WHAT LIMITS RECURSIVE REASONING MODELS: OPTIMIZATION, ARCHITECTURE AND TEST-TIME SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive reasoning models apply a small shared Transformer block many times to refine a latent state. This gives them large effective depth with few parameters and makes them strong on algorithmic tasks. Such compact solvers are natural candidates for tools that an LLM can call on narrow algorithmic subproblems. However, existing models such as HRM, TRM and URM differ in architecture, gradient propagation and training procedure simultaneously. This makes it hard to tell what drives their performance, and their optimization is still poorly understood and often unstable. In this work we address both of these gaps. First, we study these questions under a unified experimental pipeline spanning six algorithmic domains. Individual controlled ablations are performed on representative domains, while the resulting recipe is evaluated across the full suite. The study reveals a surprisingly simple recipe for stable and generalizable recursive reasoning: an intermediate gradient horizon, large physical batches and controlled updates of the recurrent state. An explicit hierarchical architecture is not needed. Second, we combine these findings into a stable 13.6M-parameter model that achieves the strongest overall performance among the evaluated recursive baselines, with particularly large gains on out-of-distribution generalization. It raises Arithmetic OOD accuracy to 71.2%, from 36.2% for the strongest baseline, while reaching 98.41% on Sudoku and 59.5% pass@2 on ARC-AGI-1. Our results show that, within the recursive architectures studied here, performance depends strongly on how recurrence is optimized and stabilized. More broadly, it shows how AI systems can be improved by optimizing their components one at a time.

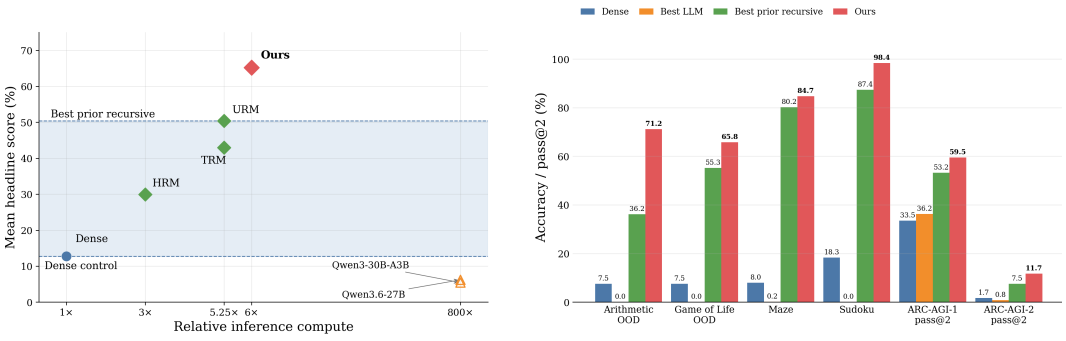


Figure 1: Recursive reasoning performance is determined by both recurrent computation and its optimization. We systematically study the factors that govern recursive models: gradient propagation through the recurrent trajectory, recurrent-state stabilization, architectural choices, and test-time computation. Our final 13.6M-parameter model combines the most effective components and achieves strong performance across algorithmic domains while improving out-of-distribution generalization.

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.

1 INTRODUCTION

Recursive reasoning models repeatedly apply a small shared network to a latent state, trading parameter count for computational depth. Recent models such as the Hierarchical Reasoning Model (HRM), Tiny Recursive Model (TRM), and Universal Reasoning Model (URM) show that this approach can solve symbolic algorithmic tasks with models of only a few million parameters (Wang et al., 2025; Jolicoeur-Martineau, 2025; Gao et al., 2025).

It is much less clear *why* these models work. HRM, TRM, and URM change recurrent architecture, gradient propagation, optimization, and regularization simultaneously, making it difficult to attribute improvements to individual design choices. Moreover, repeated application of shared parameters creates optimization challenges that differ from standard feed-forward Transformers.

We study these questions under a unified experimental pipeline. We retrain recursive baselines with matched data and evaluation, varying one architectural or optimization factor at a time. Beyond standard benchmarks (Sudoku, Maze, and ARC-AGI), we introduce Game of Life and Arithmetic with controlled out-of-distribution splits to test whether learned computation generalizes beyond the training distribution.

Our study yields three findings. First, optimization choices account for a large fraction of the performance variation observed in our study: gradient propagation has an interior optimum, and large physical batches outperform gradient accumulation at the same effective size. Second, stable recurrent computation requires controlling latent-state updates; recurrent-state stabilization is critical for reliable training. Third, additional capacity does not translate uniformly into better reasoning: explicit hierarchy, larger recurrent blocks, and additional test-time computation provide domain-dependent benefits.

Combining these findings gives a stable 13.6M-parameter recursive reasoner that achieves the strongest results among the evaluated recursive baselines on nearly all metrics, including 71.16% on Arithmetic OOD, 98.41% on Sudoku, and 59.50% pass@2 on ARC-AGI-1.

Our contributions are:

- **A controlled ablation study of recursive reasoning.** We isolate architectural and optimization choices of HRM/TRM/URM-style models under a common training and evaluation pipeline across six domains.
- **Explicit tests of algorithmic generalization.** We introduce Game of Life and Arithmetic datasets with controlled out-of-distribution splits over computational horizon, input structure, and target range.
- **A practical recipe for stable recursive optimization.** We identify an intermediate gradient horizon, large physical batches, and controlled recurrent-state updates as the key ingredients in our setting.
- **Scaling limits of the recipe.** We show that explicit hierarchy, larger recurrent blocks, and additional test-time computation provide domain-dependent benefits.

2 RELATED WORK

Latent recurrent computation. Reusing a shared transformation across multiple iterations provides additional computational depth without increasing parameter count. Looped Transformers demonstrate this principle on algorithmic reasoning and length generalization (Saunshi et al., 2025; Fan et al., 2025), while recurrent-depth language models extend it to test-time computation in larger language models (Geiping et al., 2025). Related approaches make recurrent depth adaptive across tokens (Bae et al., 2025) or perform search directly in a continuous latent program space (Macfarlane & Bonnet, 2025). These results establish latent iteration as a useful computational primitive, but leave open how such recurrence should be optimized and stabilized.

Small recursive reasoners. HRM introduced a two-timescale architecture with separate fast and slow recurrent modules, truncated credit assignment, and adaptive computation (Wang et al., 2025). TRM subsequently showed that much of this architectural structure can be removed: it shares parameters across recurrent levels, extends the differentiated portion of the trajectory, and uses weight

averaging (Jolicoeur-Martineau, 2025). URM retains the shared-module design while adding local convolutional mixing and a different truncated-backpropagation scheme (Gao et al., 2025). Subsequent analyses further question whether hierarchical structure or learned halting is essential (Ge et al., 2025; Movahedi et al., 2026), and alternative interpretations view recurrent updates as policy-improvement operations (Asadulaev et al., 2026).

These models improve along several axes simultaneously. In particular, HRM, TRM, and URM change architecture, gradient horizon, optimizer, regularization, and evaluation setup together, making it difficult to identify which factor drives their gains. Our work is complementary: rather than proposing another independent architecture, we place these choices in a shared experimental framework and vary them one at a time.

3 DATA

Prior work on recursive reasoning models evaluates on a small set of algorithmic benchmarks, most prominently Sudoku, Maze, and ARC-AGI (Wang et al., 2025; Jolicoeur-Martineau, 2025; Gao et al., 2025). We retain these benchmarks for direct comparability and add two domains, Game of Life and Arithmetic, designed to provide explicit and controllable out-of-distribution evaluation. The resulting suite contains six domains spanning constraint satisfaction, search, iterative dynamics, symbolic composition, and abstract rule induction (Figure 2).

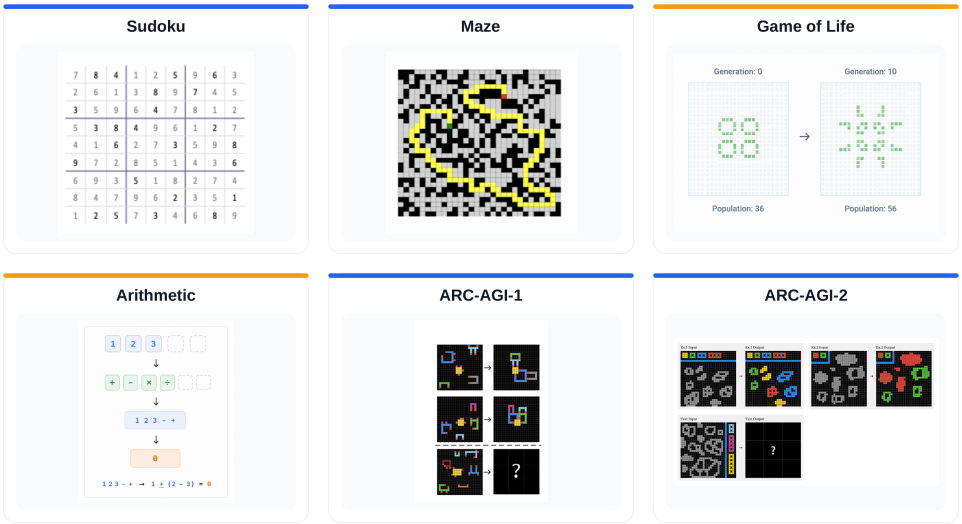


Figure 2: Evaluation domains used in our study. Sudoku, Maze, and ARC-AGI follow established recursive-reasoning benchmarks, while Game of Life and Arithmetic provide controlled out-of-distribution evaluation.

All tasks use the same interface: the input is represented as a padded token sequence, and the model predicts the complete target without intermediate supervision or chain-of-thought traces.

Sudoku. We use the `sudoku-extreme` dataset following HRM (Wang et al., 2025). Each 9×9 puzzle is flattened into 81 tokens, with the completed grid as the target. We train without symmetry-based augmentation.

Maze. We use the `maze-30x30-hard` dataset following HRM (Wang et al., 2025). Each 30×30 maze is flattened into 900 tokens, and the target is the same grid with a shortest start-to-goal path marked.

Game of Life. We generate 200,000 Conway's Game of Life trajectories from random binary grids of size up to 17×17. Each example specifies an initial configuration and a step count k, and the

target is the configuration after exactly k updates. This makes the required number of sequential rule applications directly controllable and enables evaluation of extrapolation beyond training horizons.

Arithmetic. We generate reverse-Polish expressions containing three to eight operands from $\{1, \ldots, 9\}$ and operators from $\{+, -, \times, \div\}$. Operators are masked in the input while the final value is given, and the model must recover the operator sequence. This produces a constrained combinatorial search problem with up to 4^7 candidate assignments.

ARC-AGI. We evaluate on ARC-AGI-1 and ARC-AGI-2 using the preprocessing and augmentation protocol of prior recursive models (Wang et al., 2025; Jolicoeur-Martineau, 2025; Gao et al., 2025). ARC-AGI-1 is combined with ConceptARC, while ARC-AGI-2 is kept separate. We follow the same transductive training protocol used by these models; full preprocessing and augmentation details are provided in Appendix A.

Out-of-distribution evaluation. For Game of Life, we evaluate two forms of generalization. First, 15% of initial patterns are held out entirely. Second, for patterns observed during training, we evaluate predictions 1, 2, 3, and 10 steps beyond their training horizon.

Arithmetic is split by operand multiset, so test expressions contain digit combinations absent from training. We additionally separate targets within the training value range, $[0, 101]$, from targets outside it, $[102, 201]$. Sudoku, Maze, and ARC use their standard evaluation splits.

4 Stable Recursive Reasoning

Previous recursive reasoning models change multiple factors simultaneously, including architecture, gradient propagation, and optimization procedure. We therefore isolate these factors and construct a unified recipe based on recurrent-state stabilization, credit assignment, and optimization stability. The resulting model combines shared recurrent computation with controlled state updates, intermediate gradient propagation, and recurrence-specific regularization.

The final configuration is summarized in Section 4.5, while the contribution of each component is evaluated through controlled ablations in Section 5.4.

4.1 Recursive computation

We use the nested recurrence formulation of HRM, where z_L and z_H represent low- and high-timescale recurrent states rather than separate networks, and x denotes input embeddings fixed within an Adaptive Computation Time (ACT) step. Within each high-level cycle, the low-level state is updated L_{cycles} times as

$$z_L \leftarrow L(z_L + z_H + x),$$

followed by the high-level update

$$z_H \leftarrow H(z_H + z_L).$$

The cycle is repeated H_{cycles} times. Unlike HRM, our default configuration shares the parameters of the recurrent transformations; Section 5.3 evaluates whether explicit high/low-level parameter separation provides additional benefits.

4.2 Controlled recurrent-state updates

Repeatedly applying the same transformation creates a unique optimization challenge: small errors in recurrent updates accumulate over depth. Directly replacing the recurrent state allows both the magnitude and the rate of refinement to drift along long trajectories.

Motivated by this instability, we introduce a controlled refinement mechanism consisting of update bounding, learned gating, and post-cycle normalization. We first compute the candidate update and bound its magnitude relative to the current state:

$$\tilde{z}_L = L(z_L + z_H + x), \qquad \Delta_L = \tilde{z}_L - z_L, \qquad r = \frac{\|\Delta_L\|_2}{\|z_L\|_2 + \epsilon}, \qquad \hat{\Delta}_L = \frac{\Delta_L}{\max(r/\tau, 1)}.$$

A learned gate then controls how much of the bounded update is applied:

$$\alpha = \sigma\big(w^T(z_H + z_L + x) + b\big)\,, \qquad z'_L = \mathrm{Norm}\Big(z_L + \alpha \odot \hat{\Delta}_L\Big)\,,$$

where $\odot$ denotes element-wise multiplication, with α broadcast over the hidden dimension.

Together, these mechanisms constrain the recurrent trajectory while preserving the ability of different token positions to refine at different rates.

4.3 Recurrence-consistent regularization

Because recursive models repeatedly apply the same parameters along a single latent trajectory, we reuse one dropout mask across all recurrent cycles and ACT steps rather than sampling independent masks at each application. We also add small norm-scaled Gaussian noise to the input embeddings and recurrent states to discourage brittle trajectories. Exact mask placement and regularization strengths are given in Appendix B.

4.4 Training the recursion

The model executes the full recurrent trajectory, while gradients propagate only through the final K_H high-level and K_L low-level cycles. Unless stated otherwise, we use $K_H = K_L = 2$, following the controlled study in Section 5.1. We optimize with Adam-atan2, global gradient clipping at 1, and EMA weights for evaluation, and use large physical batches rather than equivalent gradient accumulation following Section 5.2.

4.5 Final recursive reasoning recipe

The final model combines the design choices identified in our controlled study. Rather than increasing architectural complexity, we focus on stabilizing the dynamics of recurrent computation. The resulting configuration is used for all main experiments in Section 4.6.

The model uses a shared Transformer block recurrently applied to latent states. Each Adaptive Computation Time (ACT) step performs $(H_{\text{cycles}}, L_{\text{cycles}}) = (4, 2)$ recurrent updates. During training, gradients are propagated through the final $(K_L, K_H) = (2, 2)$ recurrent updates of the trajectory.

The recurrent state is updated using bounded updates, learned gating, and post-cycle normalization. Training uses recurrence-consistent dropout, relative state and embedding noise, large physical batches, gradient clipping, and exponential moving average (EMA) weights.

The final architecture is summarized in Figure 3, and the complete training recipe is given in Table 1.

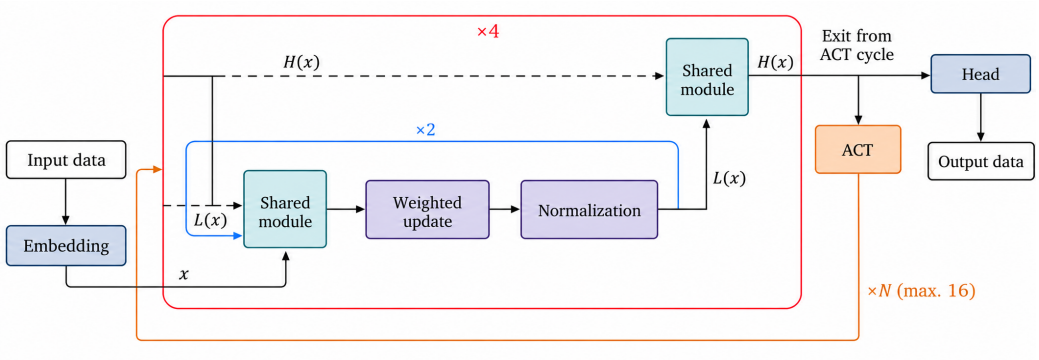


Figure 3: Overview of the final recursive reasoning architecture. A shared Transformer block repeatedly refines latent recurrent states. Controlled state updates and truncated gradient propagation stabilize the recursive trajectory.

Table 1: Comparison of recursive reasoning recipes.

Setup	**HRM**	**TRM**	**URM**	**Ours**
Parameters	27M	7M	7M	13.6M
Modules	2 $(H + L)$	1 shared	1 shared	1 shared
Layers	$4 + 4$	2	2*	4
Hidden size	512	512	512	512
Cycles (H, L)	$(2, 2)$	$(3, 6)$	$(3, 6)$	$(4, 2)$
Gradient path	Last H + last L	Last H + all L	Last H + all L	$(K_L, K_H) = (2, 2)$
State stabilization	–	–	–	Bounded + gated + norm.
Convolution	No	No	Depthwise	No**
Physical batch	768	768	768	Domain-specific; up to 8192
Train stabilization	Standard	EMA	EMA	Consistent dropout + noise

* For URM, we report the 2-layer configuration selected in our experiments.
** A convolutional variant is used only for ARC experiments and is not part of the default recipe.

4.6 Evaluation and Main Results

We report exact accuracy on all domains and pass@1/pass@2 on ARC-AGI. Adaptive halting is disabled during the main evaluation, and all examples execute the full ACT budget. For test-time scaling, we extend this budget beyond its training-time value.

We compare against HRM, TRM, and URM reconstructed in our codebase with matched data representation, tokenization, and evaluation. Each baseline retains its method-specific recurrent architecture and training recipe, while our model uses the recipe developed in 4. Thus, this comparison evaluates complete recursive-reasoning recipes rather than isolating architecture alone; architecture-specific effects are studied separately in 5. Full baseline configurations, ARC aggregation details, and inference-cost accounting are provided in Appendix C.

Table 2: Exact accuracy on the evaluation domains, in percent, from EMA weights. Arithmetic is split into the in-distribution and out-of-distribution target buckets of Section 3; Game of Life is reported on its in-range set and as the mean over its out-of-distribution sets. ARC-AGI is evaluated with pass@1 and pass@2. HRM, TRM, URM, and the dense control are retrained in our codebase; general-purpose models are evaluated from released checkpoints without fine-tuning. Best values are in bold.

Metric	Dense	Qwen3-30B-A3B	Qwen3.6-27B	HRM	TRM	URM	Ours
Arithmetic ID	78.50	0.10	0.00	92.00	**98.80**	95.60	96.34
Arithmetic OOD	7.49	0.00	0.00	13.80	36.20	28.50	**71.16**
Game of Life ID	8.17	0.00	0.00	5.00	7.11	55.42	**66.08**
Game of Life OOD	7.54	0.00	0.00	0.30	7.14	55.26	**65.80**
Maze	8.00	0.20	0.00	71.10	80.00	80.20	**84.70**
Sudoku	18.30	0.00	0.00	50.00	87.40	77.60	**98.41**
ARC-AGI-1 pass@1	30.75	25.00	30.00	32.00	40.00	49.75	**53.00**
ARC-AGI-1 pass@2	33.50	31.50	36.25	40.30	44.60	53.25	**59.50**
ARC-AGI-2 pass@1	1.67	0.83	0.00	3.33	2.50	5.83	**10.83**
ARC-AGI-2 pass@2	1.67	0.83	0.83	4.17	2.50	7.50	**11.67**

Table 2 summarizes the main comparison. Our model achieves the strongest performance among the evaluated recursive baselines on nearly all reported metrics. The largest improvement appears on Arithmetic OOD, where accuracy increases from 36.20% for the strongest recursive baseline to 71.16%.

The effect of recursion depends on the domain. On Sudoku and Maze, recursive models substantially outperform the non-recurrent control. Game of Life separates the recursive baselines: TRM remains close to the dense model, whereas URM and our model achieve higher accuracy and retain similar performance under distribution shift. ARC behaves differently: on ARC-AGI-1 the dense control is already competitive, while all systems remain limited on ARC-AGI-2.

Overall, the largest gains appear on explicit generalization tests rather than uniformly across all domains. Section 5 analyzes which optimization and architectural choices account for these differences.

5 Controlled Ablations and Scaling Limits

We isolate the main architectural and optimization choices of Section 4 under controlled comparisons. Unless stated otherwise, each experiment changes one factor while keeping the remaining training pipeline fixed. We then test whether the resulting recipe continues to improve with more model capacity, more tasks, or more inference-time computation.

5.1 Gradient Path Length

We first isolate the effect of credit assignment through the recurrent trajectory. The forward recursion is fixed at $L_{\text{cycles}} = 2$ and $H_{\text{cycles}} = 4$, while only the backward horizon is varied. Gradients propagate through the final K_H high-level cycles and, within each of them, through the final K_L low-level cycles. All eight runs use the same Arithmetic data, seed, optimizer, schedule, architecture, and forward computation.

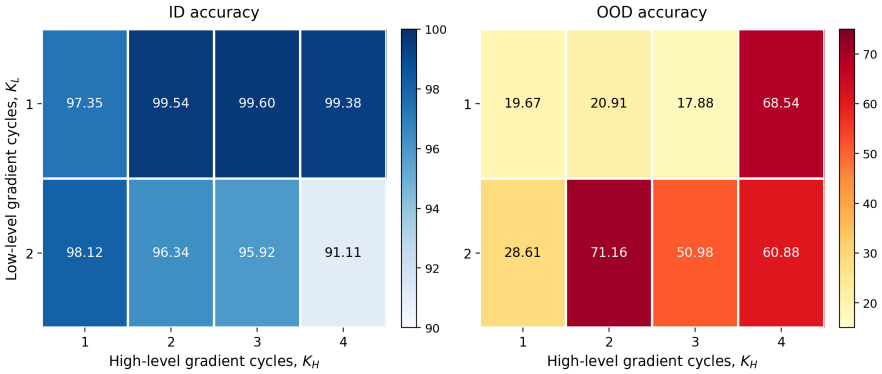


Figure 4: Exact accuracy on Arithmetic as a function of the differentiated gradient horizon. Forward recursion is fixed at $L_{\text{cycles}} = 2$ and $H_{\text{cycles}} = 4$; only (K_L, K_H) varies. Left: in-distribution accuracy. Right: out-of-distribution accuracy.

Figure 4 reveals a strongly non-monotonic relationship between gradient horizon and generalization. The intermediate setting $(K_L, K_H) = (2, 2)$ achieves the highest OOD accuracy of 71.16%, whereas $(1, 3)$ reaches the highest ID accuracy of 99.60% but generalizes poorly, obtaining only 17.88% OOD.

Importantly, this difference cannot be explained by gradient-path length alone: $(1, 3)$ and $(2, 2)$ differentiate the same number of recurrent applications but produce sharply different OOD behavior. Thus, both the extent and placement of gradient flow through the nested recurrence matter. On Arithmetic, the intermediate $(2, 2)$ setting gives the strongest out-of-distribution generalization among the tested horizons while retaining high ID accuracy.

5.2 Physical Batch vs. Gradient Accumulation

We next test whether gradient accumulation reproduces the optimization behavior of a genuinely larger physical batch. On Game of Life, we fix the effective batch size at 8192 and vary the physical batch size together with the number of accumulation steps, while keeping all other training settings fixed.

Figure 5 shows that equal effective batch size does not produce equivalent training dynamics. OOD accuracy decreases from 63.16% with a physical batch of 8192 to 37.71% with four accumulation steps, while the mean pre-clipping gradient norm increases from 0.38 to 65.46.

The effect is therefore not explained by the nominal number of examples contributing to each optimizer update. In this setting, physical batch size is an optimization parameter in its own right, and

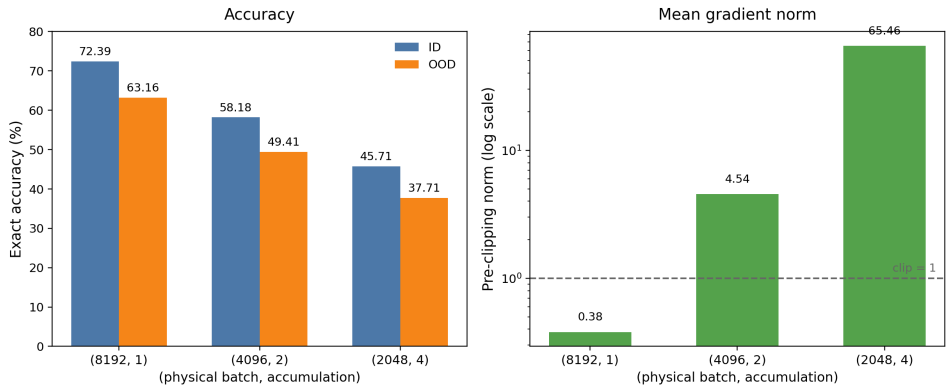


Figure 5: Effect of physical batch size versus gradient accumulation at a fixed effective batch size of 8192 on Game of Life. Configurations correspond to physical batch sizes of 8192, 4096, and 2048 with 1, 2, and 4 accumulation steps, respectively. Left: ID and OOD accuracy. Right: mean pre-clipping gradient norm.

replacing it with gradient accumulation substantially changes both stability and final accuracy. We therefore report physical and effective batch sizes separately throughout the study.

5.3 Is the Hierarchy Necessary?

HRM introduces separate high- and low-level recurrent modules, whereas TRM and URM share parameters across the two timescales. To isolate this architectural choice, we compare shared and separate recurrent modules at matched parameter count while keeping the remaining training configuration fixed.

Figure 6a shows that neither design consistently dominates across domains. Shared modules perform better on Maze and ARC-AGI-1, while separate modules provide small gains on Sudoku and Game of Life; the two configurations are nearly identical on Arithmetic.

These results indicate that explicit high/low-level parameter separation is not a universal source of improvement. A simpler shared recurrent module remains competitive across all studied domains and is preferable on several of them, suggesting that the gains of recursive reasoning do not depend on an explicit hierarchical decomposition.

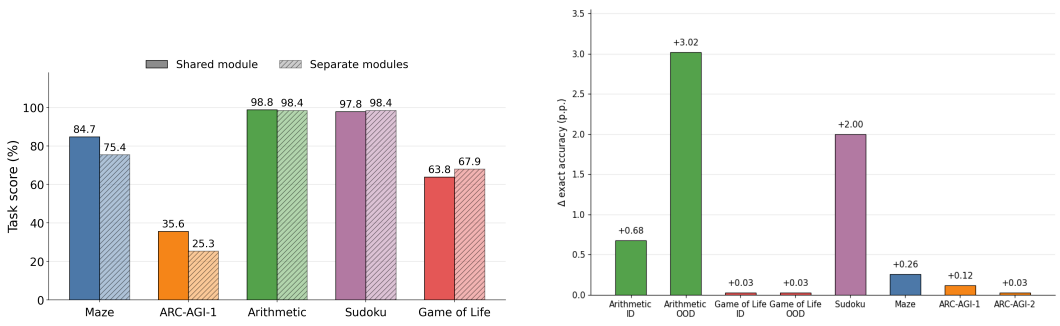
5.4 Stabilization Mechanisms

We evaluate the stabilization recipe cumulatively on Game of Life and Sudoku, focusing on both final accuracy and sensitivity to random seed. Starting from the base configuration, we progressively add gating, gradient clipping, recurrence-consistent dropout, and relative state and embedding noise. Post-cycle normalization and EMA are kept fixed throughout. Table 3 reports mean $\pm$ standard deviation over three seeds.

Table 3: Cumulative stabilization ablation on Game of Life and Sudoku. Post-cycle normalization and EMA are fixed in all configurations. Values report mean $\pm$ standard deviation over three seeds. Best values are in bold and second-best values are underlined.

Configuration	GoL ID	GoL OOD	Sudoku
Base	61.95 ± 3.32	63.87 ± 2.90	97.29 ± 1.06
+ gate	$\underline{64.24 \pm 2.59}$	$\mathbf{65.89 \pm 2.77}$	98.41 ± 0.25
+ grad clip	62.41 ± 1.56	63.87 ± 1.54	$\mathbf{98.50 \pm 0.20}$
+ dropout	61.82 ± 0.34	62.09 ± 0.41	98.36 ± 0.25
+ relative noise (final)	$\mathbf{66.08 \pm 0.11}$	$\underline{65.80 \pm 0.13}$	$\underline{98.41 \pm 0.21}$

The stabilization recipe has a pronounced effect on training reliability, especially on the more challenging Game of Life domain. The base configuration is highly seed-sensitive, with standard deviations of 3.32 and 2.90 points on the ID and OOD splits. With the complete recipe, these decrease to only 0.11 and 0.13 points, while accuracy rises to 66.08% ID and 65.80% OOD.



(a) Effect of shared versus separate recurrent modules at matched parameter count.

(b) Accuracy change from increasing the ACT budget beyond the training-time value.

Figure 6: Architectural and test-time scaling ablations. (a) Effect of sharing parameters between recurrent timescales. (b) Effect of additional recurrent computation at inference time.

Sudoku shows the same pattern in a higher-accuracy regime: variability falls from 1.06 points in the base configuration to approximately 0.2 points in the stabilized variants while mean accuracy remains near 98%. The main effect of the recipe is therefore not merely to preserve performance, but to turn a seed-sensitive recurrent optimization problem into a substantially more reproducible one. The benefit is strongest on the harder domain, where the final configuration combines both high accuracy and very low run-to-run variance.

5.5 Limits of Scaling Recursive Computation

Finally, we test whether the resulting recipe continues to improve with additional model capacity or inference-time computation. Increasing the recurrent block from four to eight layers, increasing hidden size, or adding attention heads does not yield consistent accuracy gains across the tested settings. We therefore retain the 13.6M-parameter configuration as the default model rather than increasing capacity further.

Test-time computation exhibits a similarly task-dependent pattern. Figure 6b shows that increasing the ACT budget provides the largest gains on Arithmetic and Sudoku, while improvements on Game of Life, Maze, and ARC-AGI remain limited.

Together, these results show that recursive reasoning does not improve simply by adding more parameters or more recurrent steps. Capacity and test-time compute help selectively, while the largest and most consistent improvements in our study come from how the recurrent computation is optimized and stabilized.

6 Conclusion

We presented a controlled study of recursive reasoning models, separating architectural and optimization choices that are coupled in previous systems. Across six algorithmic domains, we find that stable recursive computation depends strongly on how recurrence is trained and stabilized.

Our experiments identify three important factors. First, gradient propagation through the recurrent trajectory has an intermediate optimum: neither aggressive truncation nor full backpropagation provides the best generalization. Second, physical batch size affects optimization beyond effective batch size, with large physical batches substantially improving stability and accuracy compared with gradient accumulation. Third, controlling recurrent-state updates through bounded, gated, and normalized transformations enables reliable training.

At the same time, adding capacity does not guarantee better reasoning. Explicit hierarchy, larger recurrent blocks, and additional test-time computation provide domain-dependent rather than universal gains. These results suggest that recursive reasoning should be viewed as an optimization problem as much as an architectural one: the effectiveness of recurrence depends on how information and gradients are propagated through the recurrent trajectory.

AI USE STATEMENT

We used generative AI tools for manuscript preparation, literature discovery, and code refactoring. Generative AI was used to draft, revise, and polish parts of the manuscript and to assist with retrieval and discovery of relevant prior work. The authors independently verified all citations against the original sources and reviewed all AI-assisted text. The research questions, experimental design, experiments, measurements, analysis, and conclusions are the work of the authors.

Generative AI was also used interactively to assist with refactoring existing research code. AI-suggested changes were manually reviewed, adapted where necessary, integrated into the repository, and tested by the authors. The experimental configurations and experiments reported in the paper were controlled and executed by the authors. We reviewed all AI-assisted material and take full responsibility for the content of this paper, including its text, code, claims, and reported results.

REPRODUCIBILITY STATEMENT

The submission includes the full source code used for the experiments, together with all reported experiment configurations and the scripts used to generate the Arithmetic and Game of Life datasets. Reported experiments can be reproduced from their corresponding configuration files, which specify the model, optimization, data, evaluation, and random-seed settings. Random seeds are propagated to Python, NumPy, PyTorch, and data-loader workers.

Section 3 describes the evaluation domains and distribution shifts. The model and training procedure are described in Section 4 and Appendix B, and the evaluation protocol and baseline details are given in Section 4.6 and Appendix C. In particular, adaptive halting is disabled for the main evaluation and all examples execute the full evaluation budget.

Where results are aggregated over multiple seeds, this is stated explicitly in the corresponding table or figure caption; results obtained from a single run are reported as such.

REFERENCES

Arip Asadulaev, Rayan Banerjee, Fakhri Karray, and Martin Takac. Latent reasoning in trms is secretly a policy improvement operator, 2026.

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and Se-Young Yun. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2507.10524.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019. URL https://arxiv.org/abs/1911.01547.

François Chollet et al. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025. URL https://arxiv.org/abs/2505.11831.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2409.15647.

Zitian Gao, Lynx Chen, Yihao Xiao, He Xing, Ran Tao, Haoming Luo, Joey Zhou, and Bryan Dai. Universal reasoning model, 2025. URL https://arxiv.org/abs/2512.14693.

Renee Ge, Qianli Liao, and Tomaso Poggio. Hierarchical reasoning models: Perspectives and misconceptions, 2025. URL https://arxiv.org/abs/2510.00355.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.05171.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks, 2025. URL https://arxiv.org/abs/2510.04871.

Matthew V. Macfarlane and Clément Bonnet. Searching latent program spaces, 2025.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The ConceptARC benchmark: Evaluating understanding and generalization in the ARC domain. *Transactions on Machine Learning Research*, 2023. URL https://arxiv.org/abs/2305.07141.

Sajad Movahedi, Vera Milovanović, Shlomo Libo Feigin, Alexander Theus, Thomas Hofmann, Valentina Boeva, T. Konstantin Rusch, and Antonio Orvieto. Fixed-point reasoners: Stable and adaptive deep looped transformers, 2026. URL https://arxiv.org/abs/2606.18206. ICML 2026 Workshop.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *Proceedings of the 13th International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Pr8o5llJ1O.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi-Yadkori. Hierarchical reasoning model, 2025. URL https://arxiv.org/abs/2506.21734.

A Dataset Details

A.1 Common representation

All domains are mapped to a common sequence prediction interface. Inputs are represented as fixed-length padded token sequences, and the model predicts the complete target without intermediate supervision or chain-of-thought traces. For grid-based tasks, two-dimensional structures are flattened into token sequences. Task-specific encoding details are given below.

A.2 Sudoku

We use the `sudoku-extreme` corpus following HRM (Wang et al., 2025). Each instance is a 9×9 Sudoku board, with empty cells represented by zeros. The input board is flattened into a sequence of 81 tokens, and the target is the corresponding completed board.

We train on the full training split without symmetry-based augmentation. Thus, evaluation is performed on held-out puzzles rather than on transformed versions of a smaller underlying puzzle set.

A.3 Maze

We use the `maze-30x30-hard` corpus following HRM (Wang et al., 2025). Each instance is a 30×30 grid over the symbols $\{\texttt{wall}, \texttt{free}, \texttt{start}, \texttt{goal}, \texttt{path}\}$. The input contains the maze together with the start and goal positions. The target is the same grid with a shortest start-to-goal path marked.

Both the input and target grids are flattened into sequences of 900 tokens.

A.4 Game of Life

Generation. We generate 200,000 binary initial configurations. Grid width and height are sampled independently and uniformly from $[2, 17]$, and each cell is initially alive with probability 0.4.

Each configuration is evolved according to Conway's Game of Life using the standard B3/S23 transition rule. Evolution continues until the configuration becomes empty, enters a periodic orbit, or reaches 50 evolution steps. Patterns whose serialized representation exceeds 250 characters are discarded, resulting in encoded sequences of at most 253 tokens.

Each example consists of an initial configuration and a requested evolution step k, separated by a dedicated token. The target is the configuration obtained after exactly k applications of the Game-of-Life transition rule.

This formulation makes the required number of sequential rule applications explicit: increasing k increases the length of the computation required to obtain the target without changing the underlying transition rule.

Distribution shifts. We construct evaluation splits along two independent axes: initial configuration and computational horizon.

For pattern generalization, 15% of initial configurations are held out entirely from training. Examples derived from these configurations form the *unseen-pattern* test set.

For horizon generalization, early evolution steps from the remaining configurations are used for training. Held-out steps from the same configurations that remain within the corresponding training horizon form the *in-range* evaluation split. We additionally construct four extrapolation splits whose targets lie 1, 2, 3, and 10 evolution steps beyond the maximum horizon observed during training for that configuration.

The resulting splits therefore separate generalization to unseen initial states from extrapolation in the number of required recursive computations. Each evaluation split is subsampled to 10,000 examples.

A.5 ARITHMETIC

Generation. We generate expressions in reverse Polish notation containing between three and eight operands. Operands are sampled from $\{1, \ldots, 9\}$ and operators from $\{+, -, \times, \div\}$. Division is permitted only when it produces an exact integer result.

The resulting sequences contain at most 19 tokens. At input time, every operator is replaced by a special ? token and the final value of the expression is appended to the input. The model is trained to reconstruct the hidden operator sequence, with the loss applied only at the masked operator positions.

For an expression containing m operators, the unconstrained search space contains up to 4^m possible operator assignments. Since the longest generated expressions contain seven operators, the largest search space contains 4^7 assignments, constrained by the observed final value.

Distribution shifts. We construct train and test sets by splitting over operand multisets. The split is $80/20$, and every test expression contains a multiset of operands that does not occur in training.

Test examples are additionally partitioned according to the final expression value. Targets in $[0, 101]$ form the in-distribution evaluation set. Targets in $[102, 201]$ lie outside the range used for training and form the out-of-distribution value split.

This construction therefore combines two forms of generalization: unseen operand combinations and extrapolation to unseen target values.

A.6 ARC-AGI

We evaluate on ARC-AGI-1 (Chollet, 2019) and ARC-AGI-2 (Chollet et al., 2025). ARC-AGI-1 is combined with ConceptARC (Moskvichev et al., 2023), while ARC-AGI-2 is kept separate.

We reuse the HRM data pipeline (Wang et al., 2025), including its encoding and augmentation procedure. Each ARC grid is embedded into a 30×30 canvas with padding and explicit end-of-grid markers and is then flattened into a sequence of 900 tokens. The vocabulary contains ten color tokens and two special symbols.

Augmentation. Each puzzle is expanded into 1000 augmented examples. Augmentations compose the eight dihedral transformations of the grid with a random permutation of the color vocabulary. Training examples additionally receive a random translation within the 30×30 canvas.

At evaluation time, predictions are mapped back to the original coordinate and color systems by applying the inverse color permutation and inverse dihedral transformation and are cropped according to the end-of-grid markers.

Transductive protocol. Following prior recursive-model evaluations, few-shot demonstrations are not provided in context at inference time. Instead, each demonstration pair belonging to a task is converted into a separate supervised training example.

This also applies to tasks belonging to the evaluation split. The held-out test input itself is not used for training, but demonstration pairs from the same task are observed during training. A learned task-specific puzzle embedding is the only channel connecting these demonstrations to the held-out test input.

The resulting ARC evaluation is therefore transductive, following the protocol of the prior recursive reasoning models we compare against.

A.7 EVALUATION SPLITS

Table 4 summarizes the role of each domain in the evaluation. Sudoku and Maze use their standard held-out splits. ARC-AGI-1 and ARC-AGI-2 use the transductive protocol described above. Game of Life and Arithmetic additionally provide explicit out-of-distribution splits constructed to isolate different forms of generalization.

Table 4: Summary of the six evaluation domains. Game of Life and Arithmetic provide explicit controlled distribution shifts; the remaining domains are retained primarily for comparability with prior recursive reasoning models.

Domain	Input structure	Primary capability	Controlled OOD
Sudoku	9×9 grid	Constraint satisfaction	–
Maze	30×30 grid	Search and planning	–
Game of Life	Variable-size binary grid	Iterative dynamics	Patterns, horizon
Arithmetic	RPN expression	Symbolic composition / search	Operands, value range
ARC-AGI-1	Grid transformations	Abstract rule induction	–
ARC-AGI-2	Grid transformations	Abstract rule induction	–

B Training and Implementation Details

This appendix provides implementation details omitted from Section 4. The main text specifies the mechanisms needed to define the model; here we report the corresponding optimizer, regularization, and recurrent-carry settings.

B.1 Backbone and Optimization

The recurrent module consists of four post-norm Transformer layers with hidden size 512 and eight attention heads. The feed-forward block uses SwiGLU with expansion factor 4. We use rotary position embeddings and RMSNorm, and execute the model in bfloat16.

The high- and low-level recurrent modules share parameters, giving 13.6M trainable parameters in the configuration used for the main experiments.

We optimize the model with Adam-atan2. Training uses warm-up followed by either cosine or exponential decay, depending on the domain configuration. Peak learning rate, β_2, weight decay, and training duration are specified separately for each domain in Table 6.

Puzzle embeddings are optimized separately using distributed sign-SGD. Their learning rate is domain-specific and is reported in Table 6.

Gradients are clipped to a global norm of 1 after distributed reduction and, when applicable, after accumulation. We record the pre-clipping norm as a training-stability diagnostic.

We maintain an exponential moving average of all trainable parameters with decay 0.999, including the puzzle-embedding table.

B.2 Recurrent-State Regularization

Bounded relative update. For each token position, the relative size of the proposed state update is

$$r = \frac{\|\tilde{z}_L - z_L\|_2}{\|z_L\|_2 + \epsilon}, \qquad \epsilon = 10^{-8}. \tag{1}$$

When enabled, we use $\tau = 0.7$ and rescale the update as

$$\tilde{z}_L \leftarrow z_L + \frac{\tilde{z}_L - z_L}{\max(r/\tau, 1)}. \tag{2}$$

The scaling coefficient is detached from the computation graph.

Update gate. The gate is initialized by scaling its weight vector by 0.1 and setting its bias to zero, giving $\alpha \approx 1/2$ at initialization.

Recurrence-consistent dropout. Dropout masks are sampled when an example enters an ACT trajectory and are stored in the recurrent carry. The same masks are reused at every recurrent cycle and every ACT step until that sequence halts.

Masks are sampled separately for the QKV projection, attention residual, feed-forward residual, and feed-forward intermediate activations. Additional masks are applied to the input embeddings and the two recurrent states; the state masks are shared across token positions.

State and embedding noise. At the beginning of each ACT step during training, we perturb the input embeddings and both recurrent states according to

$$u \leftarrow u + \eta \|u\|_2 \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I), \qquad u \in \{x, z_H, z_L\}. \tag{3}$$

The norm is computed per position and detached from the computation graph. We use $\eta \in [0.003, 0.005]$ depending on the domain.

B.3 ACT and Recurrent Carry

The model uses an adaptive-computation-time Q-head with a maximum budget of 16 ACT steps, reduced to 8 for ARC-AGI-1.

Recurrent states are initialized from learned vectors. At the end of an ACT step, the resulting states are stored in the carry and detached before the next ACT step. This allows later ACT steps to continue refining the same trajectory without backpropagating through previous ACT steps.

The training objective is stablemax cross-entropy over the supervised output positions together with the two halting losses.

At evaluation time, the learned halting decision is disabled for the main accuracy comparison and every sequence executes the complete evaluation budget, as described in Section 4.6.

B.4 Domain-Specific Training Configurations

We report the domain-specific configurations used for the six final models. The recurrent architecture itself is fixed across domains, while regularization and optimization hyperparameters are adjusted to the characteristics of each task. Tables 5 and 6 summarize these differences.

Shared configuration. All models use the shared recurrent Transformer described in Section 4.5, with hidden size 512, four Transformer layers, eight attention heads, and feed-forward expansion factor four. Each ACT step performs $(H_{\text{cycles}}, L_{\text{cycles}}) = (4, 2)$ recurrent updates, and training propagates gradients through the final $(K_H, K_L) = (2, 2)$ cycles of the trajectory.

The recurrent states z_H and z_L use post-cycle normalization, recurrence-consistent dropout, and norm-scaled Gaussian perturbations. Controlled z_L updates use the relative-update threshold $\tau = 0.7$ together with the learned gate α. The remaining domain-specific choices are the dropout rates, noise scale η, and whether convolutional input mixing and the z_L gate are enabled.

Domain	d_{core}	$d_H = d_L$	Conv.	η	z_L-gate
Arithmetic	.025	.010	no	.005	yes
Sudoku	.010	.000	no	.005	yes
Game of Life	.100	.000	no	.005	yes
Maze	.010	.000	no	.005	yes
ARC-AGI-1	.025	.025	yes	.003	yes
ARC-AGI-2	.025	.025	yes	.005	yes

Table 5: **Domain-specific recurrent regularization.** d_{core} denotes dropout in the embedding, attention, residual, and feed-forward paths, while d_H and d_L denote dropout applied to the recurrent states z_H and z_L. η is the relative state and embedding noise scale. "Conv." indicates convolutional input mixing. Controlled z_L updates use $\tau = 0.7$.

Domain	Physical batch	Epochs	Eval. interval	Peak LR	Min. LR ratio	β_2	Weight decay	Puzzle LR
Arithmetic	4096	2000	50	5×10^{-4}	.01	.95	1.0	.005
Sudoku	4096	2000	50	1×10^{-4}	1.0	.95	0.1	.010
Game of Life	8192	2000	50	1×10^{-4}	.10	.95	0.1	.010
Maze	1024	54000	500	1×10^{-4}	.01	.995	0.1	.010
ARC-AGI-1	768	300000	10000	1×10^{-4}	1.0	.95	0.1	.010
ARC-AGI-2	768	300000	10000	1×10^{-4}	1.0	.95	0.1	.010

Table 6: **Domain-specific optimization configurations.** All runs use 2,000 learning-rate warmup steps, $\beta_1 = 0.9$, global gradient clipping at norm 1, and EMA weights. Puzzle embeddings use a separate learning rate ("Puzzle LR") and the same weight decay as the remaining parameters.

Optimization. The optimization procedure follows Section 4.5: all models use Adam-atan2, global gradient clipping at norm 1, and EMA weights for evaluation. The main domain-specific differences are the physical batch size, training duration, learning-rate schedule, β_2, weight decay, and puzzle-embedding learning rate. Arithmetic uses a larger peak learning rate and stronger weight decay, while Game of Life uses the largest physical batch. Maze and ARC-AGI use substantially longer schedules with smaller physical batches.

C Evaluation and Baseline Details

C.1 ARC-AGI Evaluation

ARC-AGI is scored with the standard pass@k protocol. For each test input, predictions from all augmentations are first mapped back to the original coordinate and color systems using the inverse spatial transformation and inverse color permutation. Outputs are then cropped at the end-of-grid markers and grouped by exact grid equality.

Candidate grids are ranked by vote count across augmentations, with the halting-head confidence used as a tie-break. A puzzle is counted as solved at pass@k when the ground-truth output appears among the top-k candidates for each of its test inputs. We report pass@1 and pass@2.

The main evaluation disables adaptive halting and executes the full ACT budget for every sequence, keeping batches synchronized and removing halting errors from the quality comparison. Test-time-scaling experiments instead extend the ACT budget and compare the resulting predictions.

C.2 Inference Compute

A small parameter count does not imply proportionally small inference cost in a recursive model. Weight sharing reduces parameter storage, but the shared block is still executed repeatedly.

For a model with H_{cycles} high-level cycles, L_{cycles} low-level cycles, H_{layers} high-level layers, and L_{layers} low-level layers, one ACT step requires

$$H_{\text{cycles}}\,(L_{\text{cycles}}L_{\text{layers}} + H_{\text{layers}}) \tag{4}$$

layer applications. If an example executes T ACT steps, this cost is multiplied by T.

Table 7: Inference cost of one answer relative to a single forward pass of the non-recurrent dense control. T denotes the number of ACT steps.

Model	Layers (H, L)	Cycles (H, L)	Relative compute
Dense Transformer	$8, -$	–	1
HRM	$4, 4$	$(2, 2)$	$3T$
TRM	$2, 2$	$(3, 6)$	$5.25T$
URM	$2, 2$	$(3, 6)$	$5.25T$
Ours	$4, 4$	$(4, 2)$	$6T$

Parameter sharing therefore primarily reduces parameter count rather than sequential computation: TRM and our configuration have substantially different parameter counts but similar recurrent inference cost.

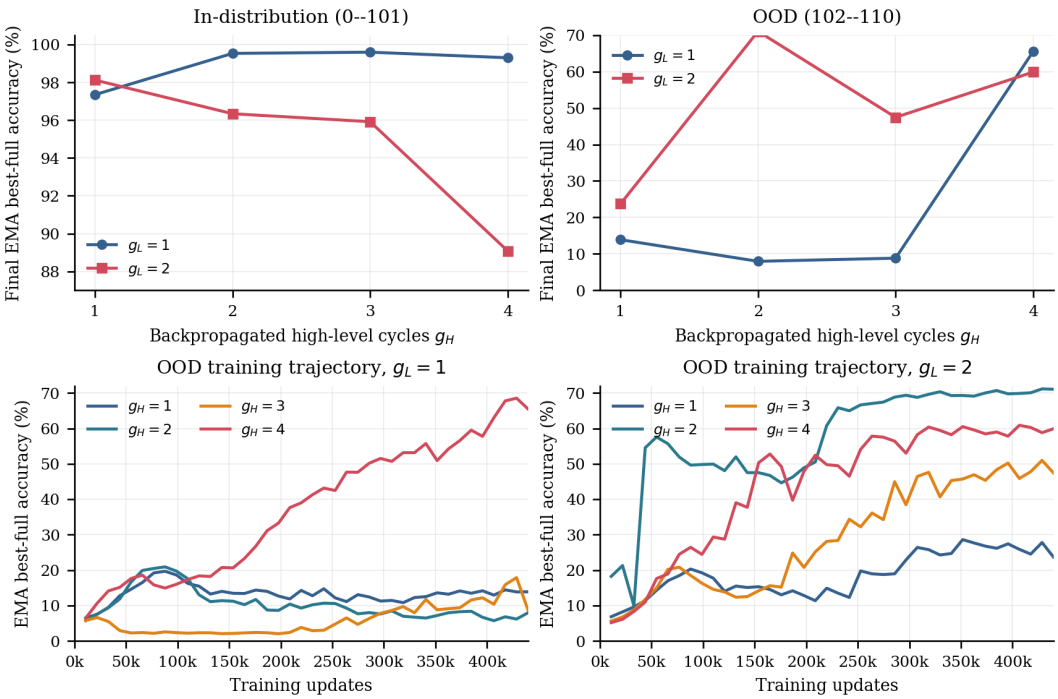


Figure 7: **Effect of the gradient-path length on Arithmetic.** Top: final EMA `best_full` exact accuracy as a function of the number of high-level cycles retained in the backward graph, for one or two retained low-level cycles. Bottom: OOD accuracy over training. The selected $(2, 2)$ model obtains the highest final OOD score.

The dense Transformer is much cheaper per answer and is therefore used as a control for recurrence rather than as a compute-matched baseline. On ARC-AGI, the augmentation-and-voting protocol introduces an additional inference cost because each puzzle is evaluated over many transformed versions. General-purpose LLMs are omitted from Table 7, since their inference cost is determined by a different autoregressive computation pattern.

D Gradient-Path Length Ablation

Our hierarchical recurrence contains four high-level cycles and two low-level cycles per high-level update. The default implementation truncates backpropagation through this recurrence. We ablate the number of terminal high-level and low-level cycles retained in the backward graph, denoted by $g_H \in \{1, 2, 3, 4\}$ and $g_L \in \{1, 2\}$, respectively. This produces a complete 2×4 set of evaluated settings. The forward recurrence, inference cost, and parameter count are unchanged; the principal architectural difference is the path through which gradients are propagated.

D.1 Setup.

All runs use the same 13.65M-parameter Arithmetic model: hidden size 512, four layers in each hierarchy, eight attention heads, four high-level cycles, two low-level cycles, and at most 16 ACT steps. They run for 2000 epochs (439,600 optimizer updates) with global batch size 4096, EMA weights, and seed 125.

Evaluation is performed every 50 epochs on the in-distribution range 0–101 and the extrapolation range 102–201. We report the EMA `best_full` exact score over the full recorded evaluation trajectory.

g_L	g_H	Final ID	Final OOD	Best OOD	Best update
1	1	97.35	13.87	19.67	87,920
1	2	99.54	7.93	20.91	87,920
1	3	**99.60**	8.80	17.88	428,610
1	4	99.31	65.53	68.54	428,610
2	1	98.12	23.71	28.61	351,680
2	2	96.34	**71.16**	**71.16**	428,610
2	3	95.92	47.46	50.98	428,610
2	4	89.08	59.88	60.88	406,630

Table 8: **EMA `best_full` exact accuracy under gradient truncation.** Accuracies are percentages. Final values are measured at 439,600 updates. "Best OOD" is the maximum over the 40 recorded checkpoints and uses the OOD split for checkpoint selection.

D.2 The selected $(2, 2)$ setting balances fitting and generalization.

The $(g_L, g_H) = (2, 2)$ run reaches 71.16% final OOD accuracy and 71.16% at its best checkpoint, the highest values in the comparison, while retaining 96.34% ID accuracy. Its final OOD score exceeds the next-best $(1, 4)$ setting by 2.62 percentage points. In contrast, several shorter paths reach 97.35%–99.60% ID accuracy but remain below 14% OOD, a pattern consistent with fitting in-range regularities without learning an extrapolating procedure. At the other extreme, extending both paths to $(2, 4)$ reduces ID to 89.08% and OOD to 59.88%, indicating that additional backpropagation depth does not translate monotonically into better optimization.

D.3 An intermediate gradient path is optimal.

The results expose a three-way trade-off. Paths that are too short provide insufficient long-range credit assignment and favor an ID–OOD generalization gap; paths that are too long can make optimization harder without improving the learned computation. The intermediate $(2, 2)$ configuration provides enough gradient reach to learn the transferable arithmetic procedure while avoiding the degradation observed at the longest setting. It is therefore the best observed balance between in-distribution fitting and extrapolative generalization, rather than simply the model with the greatest backward depth.

E Game of Life: Single-Seed Training Dynamics

This appendix examines one seed of the final Game of Life configuration to illustrate the dynamics of training. It is not used as a seed-averaged result or as an additional model comparison. The evaluation contains an in-distribution (ID) split, four out-of-distribution (OOD) horizon shifts ($+1, +2, +3, +10$ Game of Life steps), and an OOD split with unseen initial patterns.

Setup. The model has 13.65M parameters, hidden size 512, four layers in each hierarchy, eight attention heads, four high-level cycles, two low-level cycles, and at most 16 ACT steps in the training configuration. Gradients are retained through the last two high- and two low-level cycles. The model uses RoPE, a gated low-level state, stablemax cross-entropy, and dropout 0.1 on embeddings, attention, residual, and feed-forward paths.

Training uses a global batch size of 8192 across a world size of 16, peak learning rate 10^{-4} after 2000 warmup updates, minimum learning-rate ratio 0.1, weight decay 0.1, EMA parameters, and seed 536. The nominal schedule is 2000 epochs, with evaluation and checkpointing every 50 epochs. The scalar history reaches epoch 1999 and includes the complete epoch-2000 evaluation at 367,200 optimizer updates. The job was terminated after this evaluation rather than being registered as completed, so the full planned training trajectory is nevertheless available.

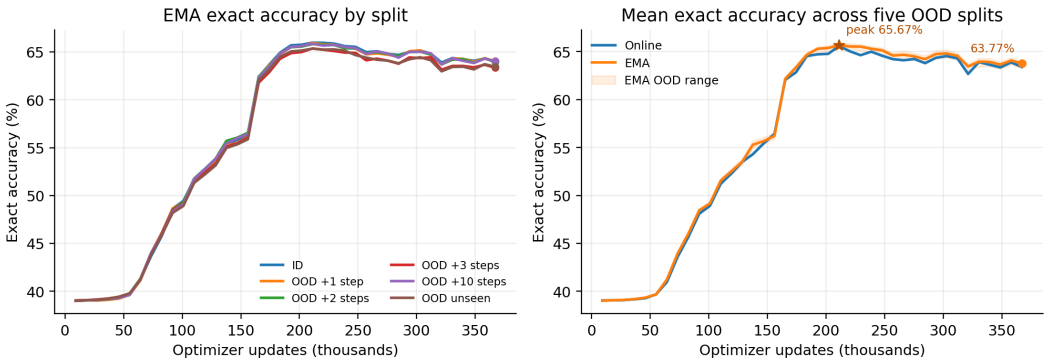


Figure 8: **Single-seed Game of Life training trajectory.** Left: EMA `best_full` exact accuracy on ID and five OOD splits. Right: the unweighted OOD mean for online and EMA parameters; shading spans the minimum and maximum EMA OOD split. Accuracy peaks at epoch 1150 and declines before the final checkpoint.

Split	Online final	Online peak	EMA final	EMA peak	EMA token acc.	Best forced step
ID	63.70	65.99	64.01	65.98	75.81	63.75 (4)
OOD +1 step	63.71	65.73	64.04	65.94	75.83	63.79 (4)
OOD +2 steps	63.65	65.73	63.94	65.83	75.92	63.67 (3)
OOD +3 steps	63.02	65.22	63.35	65.37	75.45	63.03 (3)
OOD +10 steps	63.46	65.94	64.04	65.87	75.92	63.76 (4)
OOD unseen patterns	63.12	65.34	63.46	65.36	75.69	63.12 (5)

Table 9: **Final and peak validation performance.** Values are percentages. Exact columns use the logged `best_full` exact-accuracy series; token accuracy is the EMA `accuracy` series at the final checkpoint. All EMA peaks occur at 211,140 updates (epoch 1150). "Best forced step" reports the highest single-step exact score at the final checkpoint, followed by its logged ACT step in parentheses.

The best checkpoint occurs well before the end of training. The ID EMA exact score rises from 39.04% at epoch 50 to 48.64% at epoch 500 and 64.94% at epoch 1000. It peaks at 65.98% at epoch 1150, then declines to 64.01% at epoch 2000. The five-split OOD mean follows the same trajectory: 39.02%, 48.44%, 64.67%, a peak of 65.67%, and a final value of 63.77%. Selecting the final checkpoint would therefore understate the best observed ID and OOD results by 1.97 and 1.91 percentage points, respectively.

The measured ID–OOD gap is small. At the best checkpoint, ID exact accuracy is 65.98% and the OOD mean is 65.67%, a difference of 0.31 percentage points; the OOD range is 65.36%–65.94%. At the final checkpoint, the corresponding values are 64.01% and 63.77%, with an OOD range of 63.35%–64.04%. The +3 horizon shift is the hardest final split, while +1 and +10 are marginally above ID. The differences remain small and non-monotonic.

Additional recurrence is not used for progressive refinement. At the final checkpoint, exact accuracy jumps by roughly two points over the first few forced steps. The best single-step result occurs at step 3, 4, or 5 depending on the split, after which performance is flat or drifts slightly downward. This depth profile indicates that the final model produces nearly all of its useful refinement at the start of the recurrent trajectory; the logged continuation to step 24 does not yield a second phase of computation.

Metric scope and limitations. This run is one of the seeds of the final Game of Life configuration and is shown only to characterize training dynamics. Conclusions about final model quality should rely on the seed-aggregated results in the main paper. This single trajectory does not support a causal comparison between architectures or optimization methods.

F A Negative Result: Joint Training across Four Domains

We also tested whether a single model could learn Arithmetic, Sudoku, Game of Life, and Maze jointly. To remove domain-specific differences at the input and output interfaces, we converted all

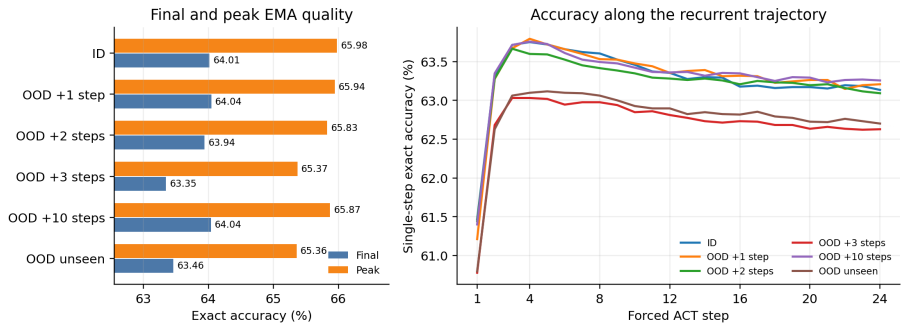


Figure 9: **Final and peak EMA `best_full` quality and recurrent-depth profile.** Left: final and peak `best_full` exact accuracy for every evaluation split. Right: EMA single-step exact accuracy at the final complete checkpoint. Most of the single-step gain occurs between forced ACT steps 1 and 2; all six curves then remain in a narrow band through step 24.

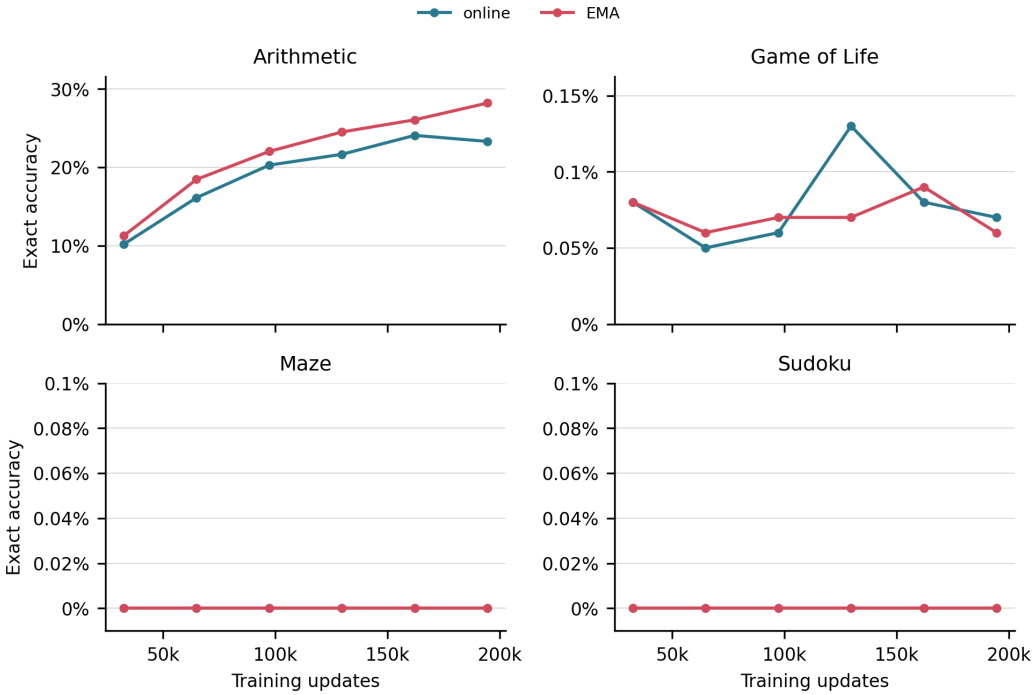


Figure 10: **Exact validation accuracy during joint four-domain training.** Online and EMA parameters are evaluated at six checkpoints. Arithmetic improves throughout training, but Game of Life remains below 0.1% exact accuracy and neither Maze nor Sudoku produces a single exact solution. The panels use different vertical scales.

four datasets to the same binary serialization and used a shared tokenizer. The resulting examples were combined into one training corpus, and no domain-specific parameters or prediction heads were introduced.

Setup. The joint model has 27.3M parameters, hidden size 512, four layers in each hierarchy, eight attention heads, and $H = L = 8$ recurrent cycles. We used a global batch size of 8192, a peak learning rate of 10^{-5}, weight decay of 1.0, a maximum of 16 ACT steps, and an exponential moving average (EMA) of the parameters. The run used seed 125. Although the configured schedule was 2000 epochs, the ClearML task was stopped after epoch 300 (195,093 optimizer updates). We therefore report all six available validation checkpoints, recorded every 50 epochs.

Domain	Final token acc.	Best exact acc.	Final exact acc.
Arithmetic	63.97	28.22	28.22
Game of Life	51.68	0.09	0.06
Maze	87.50	0.00	0.00
Sudoku	20.21	0.00	0.00

Table 10: **EMA validation metrics for joint training.** Values are percentages. "Best" is selected over the six checkpoints; final metrics are measured at 194,562 updates. Token accuracy can be high even when the full structured output is always incorrect.

Outcome. The shared model learns only a partial Arithmetic solver. Arithmetic EMA exact accuracy increases from 11.27% at the first checkpoint to 28.22% at the last. In contrast, Game-of-Life exact accuracy fluctuates between 0.06% and 0.09% for the EMA model, while Maze and Sudoku remain exactly zero at every checkpoint. The discrepancy between token and exact accuracy is especially pronounced for Maze: the final token accuracy is 87.50%, yet none of the complete predicted paths is correct. Thus the token-level loss can improve by matching many local symbols without learning the global constraints required for a valid solution.

Interpretation and limitations. Unifying the representation is therefore not sufficient, within the observed compute budget, to obtain a useful universal solver. The result is consistent with severe cross-domain optimization interference or with the joint objective being dominated by easy token-level regularities. This run does not distinguish these explanations from data-mixture imbalance, insufficient model capacity, or an inadequate optimization schedule. Moreover, the task was stopped before its nominal 2000-epoch schedule and was run with a single seed; no error is present in the recorded log that would identify why it was stopped. We consequently treat this experiment as a negative diagnostic result rather than evidence that multitask training is intrinsically ineffective, and we use separately trained domain models in the main experiments.