LOOPED ACTOR: DEPTH-RECURRENT REASONING MODELS FOR REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Looped reasoning models repeatedly apply a shared set of parameters, enabling more computation without increasing the model size. These models also support input-dependent computation by dynamically deciding when to stop looping. Motivated by the recent success of looped transformers in language modeling and reasoning, we investigate whether dynamic looping can similarly benefit sequential decision-making. We provide a complexity-theoretic motivation for this approach by showing that there exist Markov decision processes in which a state-adaptive policy achieves the optimal return with asymptotically less expected computation than any optimal fixed-runtime policy. To learn compute-adaptive policies in practice, we introduce Looped Actor, a transformer-based policy that repeatedly refines a latent representation toward a fixed point using a shared computational block. This allows the model to allocate computation adaptively by varying the number of loops based on the current state. We evaluate Looped Actor on 22 tasks across six environments, ranging from combinatorial puzzles to robotic manipulation and spanning online and offline reinforcement learning (RL) with discrete and continuous actions. Looped Actor matches or exceeds the performance of an untied baseline with 16$\times$ more parameters, with the largest gains in environments where action selection requires substantial multistep planning. For the Boxoban environment, we find that the computation allocation is structured: the number of loops increases with the number of remaining pushes and future optimal pushes become increasingly predictable from the latent state over successive loops. Together, these results highlight actor looping as a simple and efficient way to equip RL agents with adaptive computation and improve their planning capabilities.

1 INTRODUCTION

Sequential decision-making involves choices of varying difficulty: some require little effort, while others call for substantial computation. In a game, for example, choosing a strategy may require extensive planning, whereas executing it may involve a sequence of straightforward moves. A policy should therefore be able to devote more computation to difficult decisions and less to routine ones.

We theoretically motivate the computational advantage of adapting the computation of the policy to a given state using an established framework in complexity theory. More concretely, we construct a family of Markov Decision Processes (MDPs) for which we show there exist state-adaptive policies with a computational advantage over any fixed-runtime policies that grows without bound along the constructed MDP family, where both achieve optimal return. In order to learn policies with adaptive computation in practice, we propose to use looped reasoning models in the context of Reinforcement Learning (RL). Looped reasoning models naturally support such adaptation by repeatedly applying the same parameterized block and halting when reasoning is complete, enabling adaptive computation without increasing model size.

Beyond their capacity for adaptive computation, looped models have attracted growing interest as an alternative to explicit Chain-of-Thought (CoT) reasoning (Dehghani et al., 2018; Bansal et al., 2022; Saunshi et al., 2025; Hao et al., 2024; Wei et al., 2022). Early work on looped transformers focused on algorithmic tasks that decompose into subroutines and benefit from a recurrent inductive bias (Dehghani et al., 2018; Giannou et al., 2023; Kohli et al., 2026; Yang et al., 2024; Fan et al., 2025). More recently, looping has shown promise for reasoning in Large Language Models (LLMs)

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.

(Geiping et al., 2025; Zhu et al., 2025; Kapl et al., 2026). In parallel, small looped models have achieved strong performance on challenging reasoning tasks with fewer than 0.01% of the parameters of frontier models (Wang et al., 2025; Jolicoeur-Martineau, 2025; Movahedi et al., 2026; Huang et al., 2026). These results raise a natural question: can looped models similarly provide useful reasoning capabilities together with adaptive computation for sequential decision-making?

We introduce Looped Actor, a deep reinforcement learning (RL) policy that adapts the Fixed-Point Reasoning Model (FPRM) (Movahedi et al., 2026) to action selection. The actor repeatedly refines a latent representation using a shared transformer block and halts when its predicted action distribution stabilizes. This allows the policy to reason and adapt its computation to the current state without increasing the model's parameter count.

Our experiments across 22 tasks in six environments show that Looped Actor combines strong performance with computational and parameter efficiency. On Boxoban, it achieves roughly twice the success rate of an untied transformer with 16 times as many parameters, while matching this baseline's average performance across 20 offline goal-conditioned control tasks. Beyond these performance gains, we investigate how repeated computation supports planning. Probes of the actor's latent states show that future optimal box pushes become increasingly predictable over successive loops, providing evidence that the model develops representations of multistep solutions. Its allocation of computation also reflects its ability to solve a puzzle: in successful episodes, the number of loops increases with the difficulty of a state measured via the number of remaining pushes. These results suggest that Looped Actor provides a simple mechanism for combining parameter-efficient reasoning and state-dependent computation in reinforcement learning.

2 Related Work

2.1 Looped Reasoning Models

Looped architectures can increase computation without increasing the model size by repeatedly applying the same block of parameters. The first looped transformer architecture, the Universal Transformer (Dehghani et al., 2018), was specifically designed to leverage this recurrent inductive bias to effectively solve tasks that can be decomposed into smaller computational subroutines. This approach has since been extended to various algorithmic and arithmetic tasks (Giannou et al., 2023; Kohli et al., 2026; Yang et al., 2024; Fan et al., 2025). More recently, looped transformers have attracted growing interest for efficient reasoning in LLMs (Geiping et al., 2025; Zhu et al., 2025; Saunshi et al., 2025; Kapl et al., 2026), with looping enabling a new axis of test-time scaling.

A complementary line of work explores small looped transformers for challenging reasoning tasks. For example, Hierarchical Reasoning Models (HRMs) (Wang et al., 2025) and Tiny Recursive Models (TRMs) (Jolicoeur-Martineau, 2025) have been shown to outperform frontier reasoning models on benchmarks such as ARC-AGI (Chollet et al., 2024), using efficient hierarchical looped architectures with fewer than 0.01% of the parameters. These results have motivated further work on efficient reasoning with looped transformers. One framework within this line of work is attractor-based reasoning, in which a looped transformer converges to an equilibrium state that represents the solution. The Fixed-Point Reasoning Model (FPRM) (Movahedi et al., 2026) implements this approach through fixed-point dynamics. Equilibrium Reasoners (EqR) (Huang et al., 2026) and Attractor Models (Fein-Ashley & Rashidinejad, 2026) learn input-dependent attractors and use the corresponding equilibrium states as model outputs.

2.2 Iterative Computation in RL Policies

Additional computation per decision can support planning without explicit search such as Monte Carlo Tree Search (Schrittwieser et al., 2020). Early approaches embed algorithmic structure into the network, including differentiable value iteration (Tamar et al., 2016) and latent tree expansion with differentiable backups (Farquhar et al., 2017; Guez et al., 2018). Other model-based methods leverage learned abstract transition dynamics to perform lookahead rollouts (Oh et al., 2017; Silver et al., 2017) or gradient-based trajectory optimization in latent space (Srinivas et al., 2018). Thinker instead learns to plan by interacting with a learned world model before acting (Chung et al., 2023). In contrast, Looped Actor does not prescribe a particular planning algorithm or computation graph,

rather it repeatedly applies a shared computational block and learns how this recurrent computation supports action selection.

Depth recurrence replaces such templates with repeated application of a shared module. Depth-recurrent policies provide a closely related approach. Deep Repeated ConvLSTM (DRC) agents repeatedly apply a stack of ConvLSTM modules several times per observation while carrying state across environment steps (Guez et al., 2019). Subsequent work has shown that these agents exhibit signatures of emergent planning (Bush et al., 2025) and improve with additional test-time iterations (Taufeeque et al., 2024). In Sokoban, recurrence across environment steps also reduces the required within-step depth (Anokhin et al., 2026). (Ghugare et al., 2026) prove that for some tasks, policies with more computation succeed and generalize to longer horizons where compute-limited policies fail. They support this empirically with a gated recurrent block iterated a fixed number of times per step, and identify state-dependent compute allocation as an open problem. The Looped Actor addresses this by allowing the number of recurrent iterations to vary with the current state.

Adaptive computation time (Graves, 2016; Banino et al., 2021) has been studied extensively in recurrent and looped models (Dehghani et al., 2018; Zhu et al., 2025) but has seen limited use in RL. Continuous Thought Machines decouple internal ticks from the input and support certainty-based halting, with applications that include RL (Darlow et al., 2025), while Looped World Models apply an adaptive-depth looped transformer to transition prediction rather than action selection (Lu et al., 2026). To our knowledge, the Looped Actor is the first RL policy to combine a looped-transformer backbone with state-dependent iteration counts, extending the attractor dynamics of looped reasoning models to actor learning.

3 A Theoretical Motivation for Adaptive Computation

We ask whether adapting computation to the state can reduce the expected computational cost of action selection while preserving optimal return. Building on the formal framework of Ghugare et al. (2026) and using classical tools from complexity theory (Sipser, 2006), we construct a family of MDPs for which a state-adaptive implementation can achieve the same optimal return with asymptotically lower expected computation than every optimal fixed-runtime implementation under a shared description-size bound.

We represent a deterministic policy π by a single-tape Turing machine that receives a binary encoding s of an environment state and outputs a binary action. All implementations have descriptions shorter than $K_{\max}$ bits, where $K_{\max}$ is independent of input length. Let $C_\pi(s)$ denote the number of machine steps used by the chosen implementation to select an action on input s. We say that an implementation has *fixed runtime* at input length m if,

$$C_\pi(s) = B_\pi(m) \qquad \text{for every } s \in \{0,1\}^m,$$

where $\{0,1\}^m$ is the set of all binary strings of length m, and $B_\pi(m)$ is the common number of computation steps spent on each such input. An adaptive implementation may use different amounts of computation on inputs of the same length. We consider a family of one-step MDPs $(\mathcal{M}_n)_{n\geq 2}$, indexed by n. Let μ_n denote the initial-state distribution of $\mathcal{M}_n$, and let $J_n(\pi)$ denote the policy's expected return. Because each episode contains only one action, this return is its expected reward. The expected computational cost is defined by

$$\overline{C}_n(\pi) := \mathbb{E}_{s\sim\mu_n}[C_\pi(s)]\,.$$

Theorem 3.1. *There exist constants $K_0 \in \mathbb{N}$ and $c > 0$, a single adaptive implementation π_{ad}, and a family of finite one-step MDPs $(\mathcal{M}_n)_{n\geq 2}$ with binary actions and rewards in $\{0,1\}$, such that the following holds for every fixed $K_{\max} \geq K_0$ and all sufficiently large n. The initial states comprise all $(n+1)$-bit strings, each with positive probability. The adaptive implementation has description length below $K_{\max}$ and satisfies*

$$J_n(\pi_{\mathrm{ad}}) = 1, \qquad \overline{C}_n(\pi_{\mathrm{ad}}) \leq cn + 1.$$

Every optimal implementation π within the same description-size bound that has fixed runtime at length $n+1$ instead satisfies $\overline{C}_n(\pi) > n^2$. An implementation within this bound that is optimal for every n and has fixed runtime at every input length also exists. Let $\mathcal{F}_n$ denote the nonempty

set of implementations with description length below $K_{\max}$ that achieve $J_n(\pi) = 1$ and have fixed runtime at input length $n + 1$. Then,

$$\inf_{\pi \in \mathcal{F}_n} \frac{\overline{C}_n(\pi)}{\overline{C}_n(\pi_{\mathrm{ad}})} \geq \frac{n^2}{cn + 1},$$

and hence the computational advantage of adaptive computation grows without bound as $n \to \infty$.

The construction captures the simple principle that some states are easy and occur frequently while others are rare but require substantially more computation. To achieve optimal return, a fixed-runtime policy must allocate enough computation to solve the hard states and incur that cost on every input regardless of difficulty. An adaptive policy can stop earlier on easy states, achieving the same optimal return with a computational advantage that grows without bound along the constructed MDP family. This motivates the state-dependent allocation of computation implemented by Looped Actor through adaptive halting. The full proof is provided in Appendix B.1.

4 Looped Actor

We instantiate our Looped Actor with an adaptation of a recently proposed looped transformer model for reasoning, called Fixed-Point Reasoning Model (FPRM) (Movahedi et al., 2026). FPRM performs reasoning through iterative refinement of a latent representation. Given an embedded input $\mathbf{x}$, a shared, input-conditioned Transformer block f_θ repeatedly updates a latent state toward a fixed point, i.e., $\mathbf{z}^\star = f_\theta(\mathbf{z}^\star; \mathbf{x})$. Reusing the same parameters across iterations allows the model to increase its effective depth without increasing its parameter count. The model determines the number of loops itself for each input $\mathbf{x}$ by a stability criterion, which we describe below.

To support stable computation over many loops, FPRM combines pre-normalization with residual scaling at both the sublayer and looping levels. Within a stack of K Transformer blocks, each attention or feed-forward sublayer applies

$$\mathbf{h}^k = \alpha_1 \mathbf{h}^{k-1} + \beta_1 g^k_{\theta^k}\left(\mathrm{Norm}(\mathbf{h}^{k-1})\right), \qquad k = 1, \ldots, 2K.$$

Here, $g^k_{\theta^k}$ denotes the transformation performed by sublayer k, corresponding to either multi-head self-attention or a feed-forward network, with parameters θ^k shared across recurrent iterations. The representation $\mathbf{h}^{k-1}$ is the sublayer input. The input embedding is re-injected between iterations through the weighted combination $\alpha_2 \mathbf{h}^{2K} + \beta_2 \mathbf{x}$. The learnable coefficients α_1 and α_2 control retention of the residual stream and recurrent state, respectively. The remaining coefficients are coupled as $\beta_2 = 1 - \alpha_2 \alpha_1^{2K}$ and $\beta_1 = \beta_2(1 - \alpha_1)/(1 - \alpha_1^{2K})$. Under the stated boundedness assumptions and $0 \leq \alpha_1, \alpha_2 < 1$, this parameterization keeps the iterates bounded while retaining the signal-propagation benefits of pre-normalization. A depth-wise convolution at the beginning of each loop additionally supports local feature mixing for structured inputs. Together, these operations define the looped Transformer block f_θ.

While the original FPRM formulation defines halting via convergence of the relative residuals of the latent representations $\mathbf{z}^\ell$, we define halting via stabilization of the output distributions. More concretely, for a probabilistic policy predicting an output distribution $p_\ell = p_\ell(\cdot \mid s)$ conditioned on a state s at loop ℓ, e.g., Gaussian distributions with learned means or categorical distributions, we halt at the first loop satisfying

$$D_{\mathrm{KL}}(p_{\ell-1} | p_\ell) < \tau, \tag{1}$$

where D_{KL} denotes the Kullback–Leibler (KL) divergence and τ is a predefined threshold. A similar approach has been taken in Geiping et al. (2025), where the KL divergence between consecutive language token logit distributions is used as the halting mechanism.

Finally, we train Looped Actor via backpropagation through time without truncation or deep supervision, contrary to the original FPRM model; we detail the supervised readouts in Section 4.1.

4.1 Reinforcement Learning with Looped Actor

Problem setting. We formulate the RL problem as a goal-conditioned MDP described by the tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{G}, \mathcal{T}, \mathcal{R}, \gamma\}$, where $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{G}$ denote the state, action, and goal space, $\mathcal{T}$ the transition

distribution, $\mathcal{R}$ the reward function, and $\gamma \in [0, 1)$ the discount factor. Let s_t and a_t denote the state and action at time t, where actions are sampled from the policy $\pi(a_t \mid s_t, g)$. Our objective is to learn a policy that maximizes the expected return $\mathbb{E}[\sum_t \gamma^t \mathcal{R}(s_t, a_t, g)]$, either through online interaction or from a fixed offline dataset $\mathcal{D}$. When the goal is implicit in the state, we omit g. Here, we use the output distribution at the halting loop as the policy, $\pi_\theta(a_t \mid s_t, g) = p_{\ell_\theta}(a_t)$, with $\ell_\theta(s_t, g)$ given by Eq. 1. Halting introduces no additional parameters, and we train π_θ with standard RL objectives, treating halting as non-differentiable: gradients flow through the executed loops, and halted examples are frozen within a batch.

Online RL. In the online setting, we train the Looped Actor with Proximal Policy Optimization (PPO) (Schulman et al., 2017). The policy and value function share the looped block f_θ and are decoded from the latent at the halting loop, such that the critic is looped as well. Given rollouts collected with the previous policy $\pi_{\theta_{\text{old}}}$, we minimize

$$\mathcal{L}(\theta) = \mathbb{E}_t\big[- \min\big(\rho_t \hat{A}_t, \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_t\big) + c_V\big(V_\theta(s_t) - \hat{V}_t\big)^2 - c_H \mathcal{H}\big(\pi_\theta(\cdot \mid s_t)\big)\big], \quad (2)$$

where $\rho_t = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ denotes the importance ratio with clipping range ϵ, using the action probability stored at collection time, $\hat{V}_t$ the λ-return (Schulman et al., 2015), $\hat{A}_t = r_t + \gamma \hat{V}_{t+1} - V_{\theta_{\text{old}}}(s_t)$ the batch-normalized advantage formulation from Brax (Freeman et al., 2021), $\mathcal{H}$ the policy entropy, and c_V and c_H weight the value and entropy terms. Note that π_θ is recomputed during each update, so its halting loop may differ from the one employed by the behavior policy.

Offline goal-conditioned RL. In the offline setting, we build on goal-conditioned implicit Q-learning (GCIQL) (Kostrikov et al., 2021; Park et al., 2025), which samples value goals from the current state, future states of the same trajectory, or random states from $\mathcal{D}$, and actor goals from future states of the same trajectory, and assigns a reward of -1 at every step until the goal is reached. The value function V_ξ and twin critics Q_{ψ_1}, Q_{ψ_2} are MLPs trained with the expectile and temporal-difference losses of IQL, using a target critic for stability. We consider only the actor to be looped here, so that differences between models reflect only the actor architecture, and train with the DDPG+BC objective (Fujimoto & Gu, 2021; Park et al., 2024)

$$\mathcal{L}_\pi(\theta) = \mathbb{E}_{\mathcal{D}}\big[- \lambda_Q^{-1} \min\big(Q_{\psi_1}, Q_{\psi_2}\big)\big(s_t, \mu_\theta(s_t, g), g\big) - c_{\text{BC}} \log \pi_\theta(a_t \mid s_t, g)\big], \quad (3)$$

where π_θ is a Gaussian with fixed unit standard deviation, following OGBench, so that the BC term equals $\frac{1}{2}\|\mu_\theta(s_t, g) - a_t\|^2$ up to a constant, μ_θ denotes its mean, clipped to the action bounds, λ_Q normalizes the Q-values by their mean magnitude over the batch, and c_{BC} weights the behavior cloning term. The gradient of the Q-term propagates through all loops executed by the actor.

5 Experiments

The goal of the presented experiments is to isolate the benefits of looping and halting in RL policies and identify the mechanisms underlying these gains. We therefore compare Looped Actor with carefully chosen baselines, prioritizing controlled comparisons over state-of-the-art performance. Hyperparameters are provided in Appendix Section A.2.

5.1 Environments

We consider two environments in which successfully completing an episode requires multistep planning for most action decisions. We also include four environments from a recently proposed offline goal-conditioned benchmark. Together, these six environments span online and offline RL and include both continuous and discrete action spaces for a total of 22 downstream tasks. Exemplary visualizations for all six environments can be found in Appendix A.1.

Boxoban The first environment is based on Sokoban (Botea et al., 2002; Racanière et al., 2017). Each puzzle takes place on a two-dimensional board, where an agent must push boxes to goal locations. The difficulty arises from obstacles and walls, combined with the restriction that boxes can be pushed but not pulled. As a result, the agent can enter dead-end states from which recovery is impossible. Solving a Sokoban puzzle therefore requires planning multiple steps ahead to avoid such states. In this paper, we use the Boxoban version introduced by Guez et al. (2019), which fixes

the board size at 10×10. We consider the unfiltered set, comprising 900k puzzles for training, 100k for validation, and 1k for testing.

Rush hour We consider the puzzle game of Rush Hour which is played on a 6×6 board occupied by cars and trucks of length 2 and 3, respectively. The objective is to move the vehicles to clear a path for a designated car to reach the exit. This represents a very challenging planning game. In fact, its generalization to arbitrarily large boards is proven to be PSPACE-complete (Flake & Baum, 2002). In this paper, we develop an RL environment based on the rush repository[1]. Our training, validation, and test sets contain 2M, 10K, and 10K puzzles, respectively. Each game has an optimal solution length of at most 15 moves, defined as the minimum number of moves required to solve it.

OGBench We extend our study to offline goal-conditioned RL with continuous actions using four tabletop manipulation environments from OGBench (Park et al., 2025). The Cube double and Cube triple environments require rearranging cubes into a target configuration. The Puzzle 3×3 environment requires pressing buttons on a grid to reach a target color pattern, where each press toggles the pressed button and its neighbors, as in the game Lights Out (Anderson & Feil, 1998). The Scene environment requires bringing a cube, a drawer, a window, and two buttons into a target configuration, where the buttons (un-)lock the drawer and window. We train on the state-based play datasets, collected by scripted policies performing randomly chosen subtasks, and evaluate on the five goal classes provided per environment, for a total of 20 tasks. Solving these tasks requires both high-level planning of subtasks and fine-grained continuous control of the robot arm.

5.2 Model and Baseline Specifications

We limit the number of loops for our Looped Actor to a maximum of 16 iterations during training and compare it with two baselines. The **Iso-Parameters** baseline uses the same transformer block as Looped Actor but applies it only once, preserving the parameter count. The **Iso-FLOPs** baseline stacks 16 copies of this block without sharing parameters across blocks. It therefore matches the training compute of Looped Actor at the maximum loop count, but has 16 times as many parameters as either Looped Actor or the Iso-Parameters baseline. We choose the hidden dimension of the blocks to be 128, resulting in a total of 0.5M parameters for Looped Actor and Iso-Parameters, and around 8M parameters for Iso-FLOPs. Finally, we include **DRC** as a baseline on Boxoban and Rush Hour (not on OGBench as it requires a grid-structured input to its convolutional layers) because it is the closest to our approach. Since Looped Actor aims to perform multistep planning from the current state without memory, we further adapt DRC to the same stateless setting. We use the depth-recurrent variant DRC$(3, 3)$, which performed best across all environments considered in Guez et al. (2019). With approximately 1M parameters, DRC$(3, 3)$ is roughly twice the size of both Looped Actor and the Iso-Parameters baseline.

For our transformer-based models, we adopt the tokenization scheme proposed by Dong et al. (2026). Specifically, each state dimension is treated as a separate token and mapped to a latent embedding through a shared linear encoder. For goal-conditioned OGBench experiments, we append the goal tokens to the state-token sequence. For the Puzzle 3×3 environment, we instead concatenate corresponding state and goal dimensions and apply the linear encoder to each resulting pair.

5.3 Results

We report the success rate as a function of environment steps for Boxoban in Figure 1 and Rush Hour in Figure 2, showing the mean and standard deviation across three random seeds. Looped Actor substantially outperforms the Iso-FLOPs and DRC baselines despite having 2 to 16 times less parameters. The Iso-Parameters baseline shows no meaningful learning on Boxoban and, although it achieves a nonzero success rate on Rush Hour, performs substantially worse than both Looped Actor and Iso-FLOPs. Together, these results demonstrate that looped reasoning enables strong performance in environments requiring complex, multistep planning.

To assess whether Looped Actor performs well in environments that require less multistep planning than Boxoban or Rush Hour, we evaluate all Looped Actor, Iso-Parameters, and Iso-FLOPs on

[1] `https://github.com/fogleman/rush`

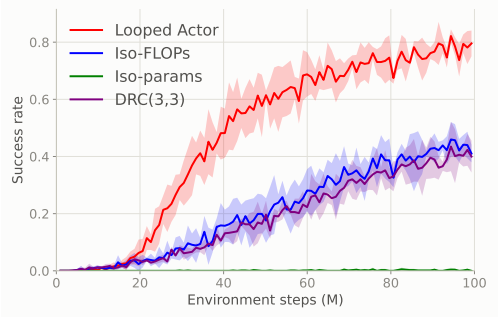


Figure 1: Boxoban success rate on unfiltered games during training for Looped Actor, Iso-FLOPs, Iso-Parameters, and DRC$(3,3)$ models.

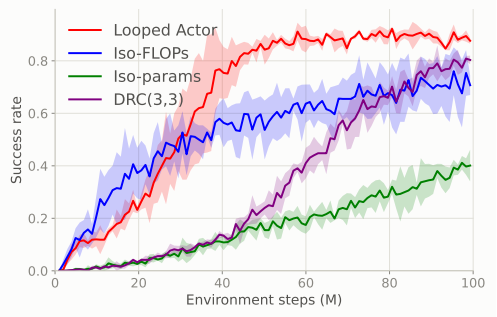


Figure 2: Rush Hour success rate during training for Looped Actor, Iso-FLOPs, Iso-Parameters, and DRC$(3,3)$ models.

the four OGBench environments considered in this paper. Table 1 reports the mean and standard deviation across five random seeds. Looped Actor ties or slightly outperforms Iso-FLOPs in all four environments. Iso-Parameters performs substantially worse, achieving zero success on at least one downstream task in three of the four environments. We provide the success rates during the full training runs in Figure 9 of the Appendix.

Together, these results show that Looped Actor matches the performance of Iso-FLOPs with substantially fewer parameters and outperforms it by a large margin in environments where most action decisions require multistep planning. Iso-Parameters consistently underperforms both models across all environments.

Multistep planning. To test whether the model builds a multistep plan over looping, we probe the latent states for knowledge of the optimal plan. We take 50k held-out Boxoban levels (25k unfiltered validation, 25k medium validation) and find the move-optimal solutions using a shortest-path solver, recording the sequences of box pushes. For each future box move in $j \in \{1,2,3,4,6,8,12,16\}$ and loop $\ell \in \{1,2,4,8,16\}$, we train a small attentive probe that reads all 101 tokens of the latent state $\mathbf{z}^\ell$ and predicts the j-th move as a (box, direction) pair among the 16 candidates. The probe is counted correct if its top choice is the j-th move of an optimal solution. As a control we train a probe on the input embedding of the puzzle $\mathbf{x}$, i.e. without any looping. Levels are split $70/10/20$ into train, validation, and test, and we report mean and standard deviation test accuracy across the three Looped Actor checkpoints on Boxoban, each checkpoint itself averaged over three seeded probes.

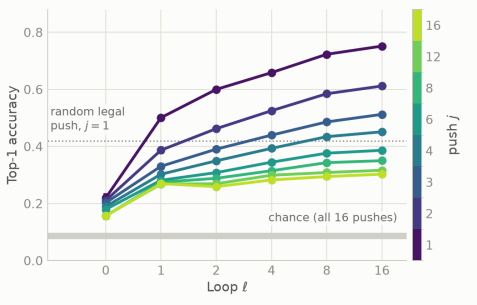


Figure 3: Accuracy in decoding the optimal plan from Looped Actor latents on Boxoban.

Figure 3 shows the optimal plan decodability as latent looping increases. Plans are present from the first loop and are refined by further loops: at every move j the decoded latent at $\ell = 1$ already beats the no-looping control ($\ell = 0$), and accuracy rises almost monotonically in ℓ. Additionally, the benefit of looping is most prominent on nearby decisions. From $\ell = 1$ to $\ell = 16$, accuracy on the next push rises from 0.50 to 0.75, on the fourth push from 0.30 to 0.45, and on the sixteenth push only from 0.27 to 0.30. Accuracy on the sixteenth push is still well above both random chance (0.0625) and the input-embedding probe (0.16). We also compare to a random guess among the pushes that are non-colliding, which scores 0.42 on the immediate push $j = 1$. The input-embedding probe (0.27) falls short of this, the latent after one loop (0.50) performs better, and accuracy increases from

Table 1: Success rate of the Looped Actor and the Iso-parameters and Iso-FLOPs baselines on OGBench tasks (%, mean ± std over 5 seeds, final checkpoint at 1M gradient steps).

Environment / task	Iso-params	Iso-FLOPs ($16 \times$ #params)	Looped Actor
Cube double	15.2 ± 7.6	66.9 ± 4.2	67.7 ± 4.6
single pnp	50.8 ± 22.0	95.6 ± 3.0	97.2 ± 1.8
double pnp 1	7.2 ± 7.6	88.8 ± 4.6	89.6 ± 13.1
double pnp 2	3.2 ± 4.6	84.8 ± 8.3	93.2 ± 5.2
swap	0.0 ± 0.0	12.8 ± 3.0	6.4 ± 2.6
stack	14.8 ± 14.1	52.4 ± 8.6	52.0 ± 7.7
Cube triple	2.7 ± 5.2	47.4 ± 7.9	50.1 ± 6.1
single pnp	8.8 ± 15.3	93.2 ± 10.0	96.8 ± 1.8
triple pnp	2.8 ± 6.3	79.6 ± 11.1	80.4 ± 20.0
pnp from stack	2.0 ± 4.5	43.6 ± 18.8	52.4 ± 18.0
cycle	0.0 ± 0.0	4.0 ± 2.4	8.0 ± 2.0
stack	0.0 ± 0.0	16.8 ± 10.4	12.8 ± 7.2
Puzzle 3×3	89.9 ± 7.5	91.4 ± 12.5	91.4 ± 7.4
task 1	99.6 ± 0.9	94.4 ± 10.4	96.8 ± 5.2
task 2	94.0 ± 5.1	90.8 ± 14.1	94.0 ± 6.9
task 3	87.2 ± 7.8	91.2 ± 12.4	86.0 ± 11.7
task 4	83.2 ± 14.2	88.4 ± 17.1	88.0 ± 6.8
task 5	85.6 ± 16.0	92.4 ± 10.4	92.0 ± 9.6
Scene play	24.2 ± 4.9	68.2 ± 4.6	69.6 ± 5.5
open	62.4 ± 18.7	99.2 ± 1.1	99.6 ± 0.9
unlock and lock	4.8 ± 2.7	96.0 ± 4.0	98.0 ± 2.4
rearrange medium	50.0 ± 7.9	90.4 ± 5.2	92.8 ± 6.6
put in drawer	3.6 ± 1.7	35.2 ± 5.0	43.6 ± 21.5
rearrange hard	0.0 ± 0.0	20.0 ± 14.8	14.0 ± 13.1
Overall average	33.0 ± 3.4	68.5 ± 4.7	69.7 ± 2.4

there. Together, these results suggest that Looped Actor spends its iterative computation sharpening local decisions while building a coarse representation of a long-horizon plan.

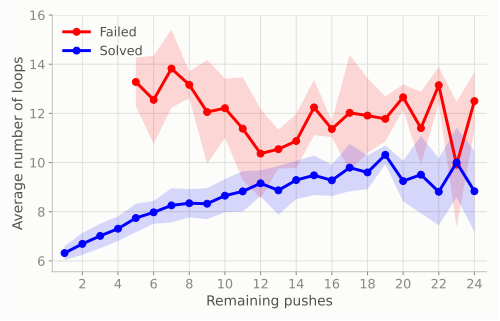


Figure 4: Loop count for individual states plotted against the corresponding number of box pushes, for both failed and solved episodes.

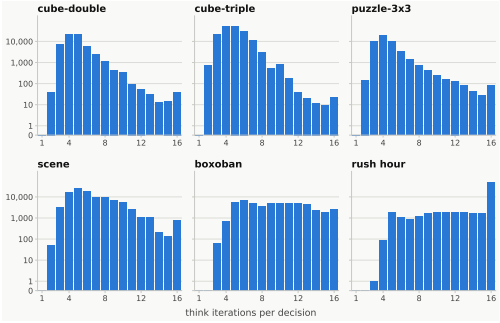


Figure 5: Histograms of loop iterations of Looped Actor for all 6 environments considered in this paper.

Learned compute allocation. Training the policy to stabilize its output distribution for each action associated with a state raises the natural question of whether the amount of compute spent to reach this stable point, i.e., the number of loops, correlates with the difficulty of the state in a meaningful way. To test this, we consider the three final Looped Actor checkpoints for Boxoban. We define the

difficulty of a particular Boxoban state as the number of remaining box pushes needed to solve the game in minimal time. The test set contains states requiring between one and 24 remaining pushes.

Figure 4 plots the average number of loops of trained Looped Actor as a function of the number of remaining pushes, for both successful and failed episodes. We can see that when the agent is able to solve an episode, the average loop count increases almost perfectly monotonically between 1 and 19 remaining pushes, ranging from 6 loops to more than 10 loops. Notably, if the agent is unable to solve an episode, the loop count does not correlate meaningfully with the difficulty of a state.

We further examine the distribution of per-state loop counts in the learned Looped Actor. Figure 5 shows these distributions across Boxoban and the five OGBench environments we consider. We can see that some states require substantially more loops. This is particularly pronounced in Boxoban and scene, which require the most planning and exhibits a long tail toward higher loop counts. Moreover, while Looped Actor is trained with a maximum loop count of 16, Figure 5 shows that the effective average depth is much smaller. Thus, Looped Actor uses both fewer parameters and fewer inference FLOPs than Iso-FLOPs.

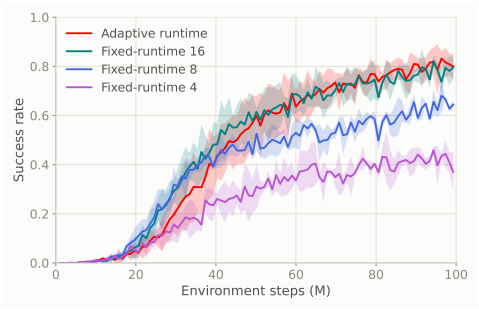


Figure 6: Adaptive vs Fixed-runtime success rates for Looped Actor on Boxoban.

To assess whether adaptive computation compromises performance, we compare our state-adaptive Looped Actor with fixed-runtime variants trained and evaluated using 4, 8, or 16 loops. Figure 6 shows success rates throughout training. Using fewer than 16 loops substantially reduces performance, whereas the fixed-runtime variant with 16 loops matches our state-adaptive Looped Actor. Yet more than 95% of trajectories in the adaptive model halt before reaching 16 loops, yielding a speedup of over 60% relative to the fixed-runtime counterpart. These results demonstrate the computational benefits of adaptive computation: the policy devotes additional computation to states that require it, avoiding the cost of running all 16 loops at every state.

6 Conclusion

We introduced Looped Actor, a depth-recurrent reasoning model policy that allocates computation adaptively across states by repeatedly applying a shared transformer block until its action distribution stabilizes. We provided a theoretical motivation for state-dependent computation, constructing a family of MDPs in which an adaptive policy achieves optimal return with asymptotically lower expected computation than any optimal fixed-runtime policy under the same description-size constraint. Empirically, Looped Actor matches or exceeds a larger untied transformer (with 16 times as many parameters) across 22 tasks spanning online and offline RL. While the largest improvements are on problems requiring multistep planning, we also show that Looped Actor retains competitive performance on high dimensional continuous control tasks. Through latent probes, we show that successive loops progressively refine representations of future optimal actions. In addition, we showed that the number of loops increases with planning difficulty in successful multistep planning episodes. Finally, we showed empirically that loop counts vary widely across states, highlighting the potential computational savings of adapting computation to each state rather than applying a large, fixed budget uniformly. Together, these results suggest that Looped Actor offers a simple yet effective approach to equip RL agents with adaptive computation and improve their planning capabilities.

Several limitations and open questions remain. Although our theoretical construction in Section 3 shows that compute-adaptive policies can be substantially more efficient than fixed-time policies, there is no guarantee that Looped Actor learns a compute-optimal policy in practice. Developing approaches that further improve the computation–reward Pareto frontier remains an important direction for future work. Our formulation also focuses on within-step computation without carrying internal reasoning across environment steps, leaving open how adaptive depth should interact with memory and partial observability.

AI USE STATEMENT

We used generative AI tools to assist with implementing the experiments, refining the proof's presentation, addressing gaps in its argument, and polishing the manuscript's language. The authors reviewed all AI-assisted work, including the experimental implementation, and checked and verified every step of the proof. We take full responsibility for the final content of this work, including all text, claims, proofs, code, and other artifacts produced with generative AI assistance.

REPRODUCIBILITY STATEMENT

To facilitate full reproduction of our experimental results, we provide the complete source code implementing our experiments in the supplementary materials. All experiments use fixed random seeds to support reproducibility. In addition, we conduct runs across multiple seeds to assess the robustness of our findings to randomness in the experimental procedure.

REFERENCES

Marlow Anderson and Todd Feil. Turning lights out with linear algebra. *Mathematics magazine*, 71 (4):300–303, 1998.

Ivan Anokhin, Johan Obando-Ceron, Irina Rish, and Sebastian Risi. Temporal recurrence favors fewer layers. *arXiv preprint arXiv:2609.12531*, 2026.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end algorithm synthesis with recurrent networks: Logical extrapolation without overthinking. *Advances in Neural Information Processing Systems*, 35:20232–20242, 2022.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In *International conference on computers and games*, pp. 360–375. Springer, 2002.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

Thomas Bush, Stephen Chung, Usman Anwar, Adrià Garriga-Alonso, and David Krueger. Interpreting emergent planning in model-free reinforcement learning. In *International Conference on Learning Representations*, volume 2025, pp. 82115–82197, 2025.

Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report. *arXiv preprint arXiv:2412.04604*, 2024.

Stephen Chung, Ivan Anokhin, and David Krueger. Thinker: Learning to plan and act. *Advances in Neural Information Processing Systems*, 36:22896–22933, 2023.

Luke Darlow, Ciaran Regan, Sebastian Risi, Jeffrey Seely, and Llion Jones. Continuous thought machines. *Advances in Neural Information Processing Systems*, 38:21548–21594, 2025.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Perry Dong, Kuo-Han Hung, Alexander Swerdlow, Dorsa Sadigh, and Chelsea Finn. Tql: Scaling q-functions with transformers by preventing attention collapse. *arXiv preprint arXiv:2602.01439*, 2026.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *International Conference on Learning Representations*, volume 2025, pp. 14502–14520, 2025.

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

Jacob Fein-Ashley and Paria Rashidinejad. Solve the loop: Attractor models for language and reasoning. *arXiv preprint arXiv:2605.12466*, 2026.

Gary William Flake and Eric B Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1-2):895–911, 2002.

C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

Scott Fujimoto and Shixiang Shane Gu. A minimalist approach to offline reinforcement learning. *Advances in neural information processing systems*, 34:20132–20145, 2021.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2025.

Raj Ghugare, Michał Bortkiewicz, Alicja Ziarko, and Benjamin Eysenbach. On the role of computation in reinforcement learning. *arXiv preprint arXiv:2602.05999*, 2026.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. In *International conference on machine learning*, pp. 1822–1831. PMLR, 2018.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy Lillicrap. An investigation of model-free planning. In *International conference on machine learning*, pp. 2464–2473. PMLR, 2019.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Benhao Huang, Zhengyang Geng, and Zico Kolter. Equilibrium reasoners: Learning attractors enables scalable reasoning. *arXiv preprint arXiv:2605.21488*, 2026.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.

Ferdinand Kapl, Emmanouil Angelis, Kaitlin Maile, Johannes von Oswald, and Stefan Bauer. From growing to looping: A unified view of iterative computation in llms. *arXiv preprint arXiv:2602.16490*, 2026.

Harsh Kohli, Srinivasan Parthasarathy, Huan Sun, and Yuekun Yao. Loop, think, & generalize: Implicit reasoning in recurrent-depth transformers, 2026. URL https://arxiv.org/abs/2604.07822.

Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. *arXiv preprint arXiv:2110.06169*, 2021.

Hongyuan Adam Lu, Z. L. Victor Wei, Qun Zhang, Jinrui Zeng, Bowen Cao, Lingwei Meng, Mocheng Li, Zezhong Wang, Haonan Yin, Naifu Xue, Minyu Chen, Cenyuan Zhang, Zefan Zhang, Hao Wei, Jiawei Zhou, Haoran Xu, Hao Yang, Ronglai Zuo, Tongda Xu, Yonghao Li, Jian Chen, Hebin Wang, Zeyu Gao, Yang Li, Wei Zhao, Qimin Zhong, Siqi Liu, Yumeng Zhang, Leyan Cui, Zhangyu Wang, and Wai Lam. Looped world models. *arXiv preprint arXiv:2606.18208*, 2026.

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. *arXiv preprint arXiv:2606.18206*, 2026.

Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *Advances in neural information processing systems*, 30, 2017.

Seohong Park, Kevin Frans, Sergey Levine, and Aviral Kumar. Is value learning really the main bottleneck in offline rl? *Advances in Neural Information Processing Systems*, 37:79029–79056, 2024.

Seohong Park, Kevin Frans, Benjamin Eysenbach, and Sergey Levine. Ogbench: Benchmarking offline goal-conditioned rl. In *International Conference on Learning Representations*, volume 2025, pp. 94937–94982, 2025.

Sébastien Racanière, Theophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *International Conference on Learning Representations*, volume 2025, pp. 14855–14881, 2025.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588 (7839):604–609, 2020.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3191–3199. PMLR, 06–11 Aug 2017.

Michael Sipser. *Introduction to the Theory of Computation*. Thomson course technology Boston, 2006.

Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal planning networks: Learning generalizable representations for visuomotor control. In *International conference on machine learning*, pp. 4732–4741. PMLR, 2018.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in neural information processing systems*, 29, 2016.

Mohammad Taufeeque, Philip Quirke, Maximilian Li, Chris Cundy, Aaron David Tucker, Adam Gleave, and Adrià Garriga-Alonso. Planning in a recurrent neural network that plays sokoban. *arXiv preprint arXiv:2407.15421*, 2024.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical Reasoning Model, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. 2024.

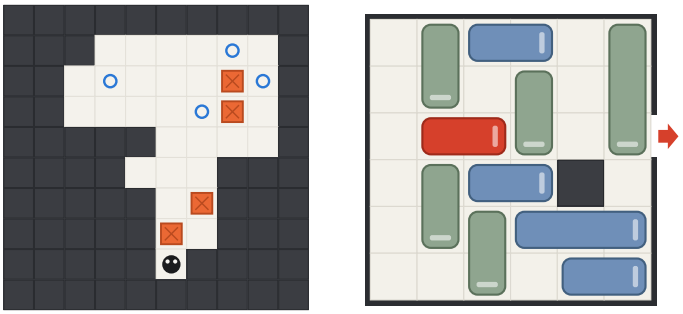
Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, Lu Li, Jiajun Shi, Kaijing Ma, Shanda Li, Taylor Kergan, Andrew Smith, Xingwei Qu, Mude Hui, Bohong Wu, Qiyang Min, Hongzhi Huang, Xun Zhou, Wei Ye, Jiaheng Liu, Jian Yang, Yunfeng Shi, Chenghua Lin, Enduo Zhao, Tianle Cai, Ge Zhang, Wenhao Huang, Yoshua Bengio, and Jason Eshraghian. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

Supplementary Material for:
Looped Actor: Depth-Recurrent Reasoning Models for Reinforcement Learning

A Experimental Details

All experiments were run on NVIDIA H100 and B200 GPUs. Both the environments and the training code were implemented in JAX (Bradbury et al., 2018).

A.1 Visualization of Environments



(a) Example of a Boxoban game. (b) Example of a Rush Hour game.

Figure 7: Examples of the two game environments.

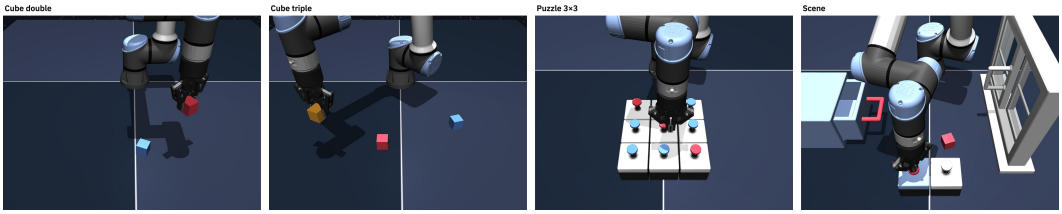


Figure 8: OGBench environments for Cube double, Cube triple, Puzzle 3x3, and scene.

A.2 Hyperparameters

We chose the same fixed hyperparameters for all Transformer models considered in this paper. Table 2 provides the model hyperparameters, while Table 3 shows the training hyperparameters.

Hyperparameter	OGBench	Boxoban	Rush Hour
Layers per iteration	2	2	2
Model width	128	128	128
Attention heads	4	4	4
MLP expansion ratio	4	4	4
Grid convolution	No	Yes	Yes
Train iterations (min/max)	2/16	2/16	2/16
Halting threshold τ	10^{-3}	10^{-3}	10^{-3}

Table 2: Model hyperparameters.

Hyperparameter	OGBench	Boxoban / Rush Hour
Algorithm	GCIQL	PPO
Optimizer	Adam	Adam
Learning rate	3×10^{-4}	10^{-4}
Gradient clipping (global norm)	1.0	1.0
Batch / minibatch size	1024	1024
Training budget	10^6 gradient steps	10^8 environment steps
Discount γ	0.99	0.99
Parallel environments	–	1024
Rollout steps per environment	–	64
Epochs per rollout	–	4
GAE λ	–	0.95
PPO clip ϵ	–	0.3
Entropy coefficient	–	0.01
Value loss coefficient	–	0.25
IQL expectile	0.9	–
Target EMA τ	0.005	–
DDPG+BC α	1.0	–

Table 3: Optimization hyperparameters. Boxoban and Rush Hour share all settings listed here.

B PROOFS

B.1 PROOF OF THEOREM 3.1

Proof idea. We construct a decision problem with frequent easy states and rare states on which the policy must solve a computationally demanding problem. The key lower bound comes from one state for each candidate policy: a state containing that policy's own program description. An optimal policy cannot finish within n^2 steps on this state. If its runtime is fixed, it must therefore spend more than n^2 steps on every state of the same length. An adaptive policy can instead stop quickly on the frequent easy states.

Computational conventions. Policy implementations are deterministic single-tape Turing machines that halt on every finite binary input. Fix an explicit, polynomial-time parsable binary encoding of finite transition tables, with state and tape-symbol identifiers written in binary. Let $\langle M \rangle$ denote the description of machine M, and let M_π denote the machine implementing policy π. The size constraint $|\langle M_\pi \rangle| < K_{\max}$ counts the program description, excluding its input and working tape contents. Runtime counts all executed machine transitions.

Proof. **1. Define the correct action on the hard branch.** We adapt the diagonalization in the proof of Theorem A.1 of Ghugare et al. (2026). Let U be a universal simulator: given a program description and an input, it executes that program on that input.

For $n \geq 2$ and $x \in \{0,1\}^n$, check whether x has the form

$$x = 1^\ell 0 p 0^k, \qquad |p| = \ell, \qquad k = n - 2\ell - 1 \geq 0,$$

where p is a valid machine description. Here juxtaposition means concatenation, 1^ℓ means ℓ consecutive ones, and 0^k means k consecutive zeros. The prefix specifies the length of the program description, and the trailing zeros pad the string to length n.

For a valid encoding, use U to execute the program described by p on input $1x$, stopping after at most n^2 steps of the encoded program. Here p specifies the instructions, while $1x$ is the complete state on which those instructions are executed. Define

$$h_n(x) = \begin{cases} 1 - a, & \text{if the program outputs } a \in \{0,1\} \text{ within } n^2 \text{ steps,} \\ 0, & \text{otherwise, including invalid encodings.} \end{cases}$$

Thus h_n always returns a binary answer. It deliberately disagrees with the encoded program whenever that program finishes within the cutoff. The cutoff counts steps of the encoded program; evaluating h_n can take more actual steps because simulation has overhead.

Using padded configurations and a prescribed sequence of full tape sweeps, parsing and simulation can follow a schedule determined solely by n. All n^2 simulation rounds are executed, retaining the recorded result after simulated halting. This yields a polynomial-time implementation whose actual runtime is identical on all length-n inputs.

2. Construct two implementations of the same policy. Write the observed state as $s = bx$, where $b \in \{0,1\}$ and $x \in \{0,1\}^n$. The adaptive implementation π_{ad} scans the input and uses b to select a branch: if $b = 0$, it outputs 0 immediately after the scan; if $b = 1$, it computes and outputs $h_n(x)$. Consequently, for some constant $c > 0$ and an integer-valued polynomial $q(n) \geq n^2$,

$$C_{\pi_{\mathrm{ad}}}(0x) \leq cn, \qquad C_{\pi_{\mathrm{ad}}}(1x) \leq q(n) \quad \text{for every } x \in \{0,1\}^n,\ n \geq 2. \tag{4}$$

The easy branch only scans the input while remembering b. The polynomial q bounds all work on the hard branch, including parsing and simulation overhead; a sufficiently large integer multiple of $(n+1)^d$ suffices for a sufficiently large fixed integer d.

The fixed-runtime implementation π_{fix} computes $h_n(x)$ on every input, regardless of b, using the fixed schedule above. It then outputs 0 if $b = 0$, discarding the computed answer, and outputs $h_n(x)$ if $b = 1$. Make this final selection take the same number of steps in either case. This program selects the same actions as π_{ad}, but its runtime depends only on input length. For inputs shorter than three bits, both machines perform a fixed scan and output 0.

Both programs have constant descriptions independent of n. Choose

$$K_0 = 1 + \max\bigl\{|\langle M_{\pi_{\mathrm{ad}}}\rangle|, |\langle M_{\pi_{\mathrm{fix}}}\rangle|\bigr\}.$$

Then both descriptions are shorter than every $K_{\max} \geq K_0$. This particular fixed-runtime implementation establishes existence; the lower bound below will apply to every optimal fixed-runtime implementation, regardless of its algorithm.

3. Define the rewards and make the hard branch rare. The initial state space is $\mathcal{S}_n = \{0,1\}^{n+1}$. At state bx, action 0 is rewarded when $b = 0$, while action $h_n(x)$ is rewarded when $b = 1$:

$$r_n(bx, a) = \begin{cases} \mathbf{1}\{a = 0\}, & b = 0, \\ \mathbf{1}\{a = h_n(x)\}, & b = 1. \end{cases}$$

Here $\mathbf{1}\{E\}$ is one if E holds and zero otherwise. After the action, the MDP transitions deterministically to a separate terminal state $\perp$, with no further action or reward. Thus $h_n(x)$ specifies the correct action, not the reward itself. Both constructed implementations always choose correctly and achieve optimal return 1.

Sample x uniformly from $\{0,1\}^n$ and independently choose $b = 1$ with probability $1/q(n)$. Every initial state has positive probability:

$$\mu_n(1x) = \frac{2^{-n}}{q(n)}, \qquad \mu_n(0x) = 2^{-n}\left(1 - \frac{1}{q(n)}\right).$$

Combining these probabilities with equation 4 gives

$$\overline{C}_n(\pi_{\mathrm{ad}}) \leq \left(1 - \frac{1}{q(n)}\right) cn + \frac{1}{q(n)} q(n) \leq cn + 1.$$

The hard branch can be expensive, but its contribution to expected computation is at most one because it occurs with probability $1/q(n)$. The two programs, q, and the MDP family are all fixed independently of $K_{\max}$.

4. Use one state to lower-bound the common runtime. Fix $K_{\max} \geq K_0$ and $n \geq 2K_{\max} + 1$. Consider any optimal fixed-runtime implementation $\pi \in \mathcal{F}_n$. It may use any algorithm; it need not evaluate h_n by simulation. Let $p = \langle M_\pi \rangle$ be its own program description and $\ell = |p| < K_{\max}$. The choice of n ensures that this description fits inside the string

$$x_\pi = 1^\ell 0 p 0^{\,n-2\ell-1}.$$

The corresponding state $1x_\pi$ belongs to $\mathcal{S}_n$ and has positive probability.

Suppose, for a contradiction, that the common runtime satisfies $B_\pi(n+1) \leq n^2$. On state $1x_\pi$, the policy then finishes within n^2 steps and outputs some action $a = \pi(1x_\pi)$. To determine the

rewarding action at this state, the rule h_n extracts the encoded program and runs it on $1x_\pi$. The encoded program is exactly M_π, and its input is exactly the state on which we are evaluating π. Therefore the simulation finishes within the cutoff with the same output a, giving

$$h_n(x_\pi) = 1 - a.$$

The policy chooses a, while the rewarding action is $1 - a$, so its reward on this state is zero. Because the state has positive probability,

$$J_n(\pi) \leq 1 - \mu_n(1x_\pi) < 1,$$

contradicting optimality. Hence $B_\pi(n+1) > n^2$.

Only this one state was needed to establish the lower bound. Since π has fixed runtime, the same bound applies on every other input of length $n+1$, including the easy states. Thus $\overline{C}_n(\pi) = B_\pi(n+1) > n^2$ for every $\pi \in \mathcal{F}_n$. The implementation π_{fix} makes this class nonempty. Combining the lower bound with the adaptive upper bound proves

$$\inf_{\pi \in \mathcal{F}_n} \frac{\overline{C}_n(\pi)}{\overline{C}_n(\pi_{\text{ad}})} \geq \frac{n^2}{cn+1}.$$

The reward rule is fixed throughout this argument: it always examines the program encoded in the observed state. Only the state $1x_\pi$ used to demonstrate failure depends on the candidate policy. □

C Additional Results

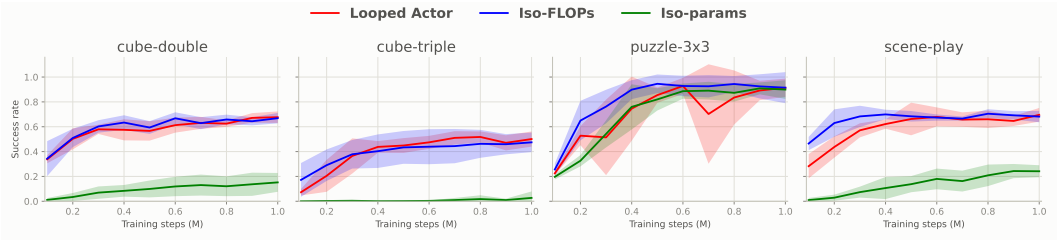


Figure 9: Success rate of Looped Actor, Iso-FLOPs, and Iso-Parameters models on all four OG-Bench environments considered in this work during training (mean and standard deviation over 5 seeds).