TEMPORAL RECURRENCE FAVORS FEWER LAYERS

Anonymous authors
Paper under double-blind review

ABSTRACT

In streaming tasks, recurrent models can carry latent computation across time, allowing each update to build on representations produced earlier. This raises a basic question: once temporal recurrence provides sequential computation across steps, how much depth is still needed within each step? Prior work has shown that recurrence can make shallow models competitive. We instead study this question as a compute-allocation problem, varying within-step depth, expert width, and the number of parallel experts per layer across several compute budgets. For each budget, we compare the best observed recurrent and non-recurrent allocations and the performance they achieve under approximately matched per-step computation. Across Sokoban and autoregressive FineWeb language modeling, we find that temporal recurrence shifts the best observed compute allocation toward substantially fewer layers, with comparable or better performance.

1 INTRODUCTION

Across many deep learning tasks, models benefit from progressively refining their internal representations through a sequence of latent transformations (He et al., 2016; Brown et al., 2020; Wang et al., 2026; Dehghani et al., 2019). This refinement is commonly implemented by stacking layers, allowing later layers to build on intermediate states produced by earlier ones. Depth therefore controls one important source of sequential computation available to a model.

In natural *streaming* tasks, sequential computation can also accumulate across time. The model is repeatedly applied as the task unfolds, for example, across tokens in language modeling, frames in video, or observation steps in reinforcement learning. A recurrent model can carry a latent state across these steps, allowing each update to build on computations performed at earlier steps. Recurrence can therefore do more than preserve information from the past: it can also continue refining representations or plans for future predictions and actions. This temporal recurrence is different from depth recurrence, as used in looped Transformers, where shared blocks are repeatedly applied on the same input before producing an output (Dehghani et al., 2019; Darlow et al., 2026; Jolicoeur-Martineau, 2025; Geiping et al., 2026). Here, computation continues across successive token or environment steps as new inputs arrive.

Whether sequential computation can accumulate across external steps depends on the structure of the task. It can do so when several consecutive steps contribute to the same broader problem, allowing computation performed at one step to remain useful later. One favorable setting is when relatively easy predictions or actions come before a harder one: an agent may refine a plan while walking through a corridor, and a language model may continue refining its representation of a passage while generating locally predictable tokens. In contrast, if each step presents a new independent problem that must be solved immediately, earlier computation has no value for later steps.

Prior work demonstrates that models can learn useful computation across external steps. State-tracking studies train models to predict the current state after each input operation (Merrill et al., 2024). On difficult permutation-composition tasks, single-layer recurrent models succeed across the tested sequence lengths, while Transformers require increasing depth to maintain accuracy. Evidence also comes from reinforcement learning, where recurrent agents learn planning-like behavior in fully observed Sokoban environments (Guez et al., 2019). Subsequent analysis shows that their internal plans develop *across* environment steps and causally influence their actions (Bush et al., 2025).

Our code is available at Anonymous.

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.

This raises the central question of the paper: *How much within-step depth is still needed once temporal recurrence is available?* Prior work has shown that recurrence can make shallow models competitive (Merrill et al., 2024; Guez et al., 2019; Fan et al., 2020; Oncescu et al., 2026). We instead formulate this question as a compute-allocation problem, comparing similar recurrent and non-recurrent models. Let S denote the number of sequential stages within that step, and let P denote the parallel compute performed at each stage. The total compute per step is then approximately $C \approx S \cdot P$. At fixed C, allocations with fewer sequential stages spend more compute in parallel at each stage. We ask whether temporal recurrence changes the best allocation: do recurrent models require a smaller S than non-recurrent models to achieve comparable performance?

To realize this comparison, we use recurrent and non-recurrent models with the same three allocation axes, illustrated in Fig. 1. The within-step depth L controls the number of successive stages (layers) S. Within each stage, multiple experts are evaluated in parallel and their outputs are combined, allowing P to be increased in two ways: by adding more experts E, or by increasing the hidden dimension d of each expert. The recurrent variant carries a latent state across external steps, while the non-recurrent variant does not. Varying L, E, and d under matched work lets us compare how much within-step depth remains useful with and without temporal recurrence.

Our experiments support two main conclusions:

- **Temporal recurrence shifts the best compute allocation toward fewer layers.** Across Sokoban and streamed FineWeb, the best observed recurrent allocations use substantially fewer layers per step than the best non-recurrent allocations, while achieving comparable or better performance under approximately the same compute budget.
- **Recurrence and parallel capacity play different roles.** Increasing per-layer expert count benefits both recurrent and non-recurrent models similarly, while gains from additional layers saturate earlier with recurrence. Recurrence therefore reduces the performance cost of using fewer layers, while expert count and width provide useful ways to spend the reallocated compute.

More broadly, we hypothesize that temporal recurrence will shift the best compute allocation toward shallower within-step updates in other streaming settings where successive steps contribute to a shared problem and carried state can support continued refinement.

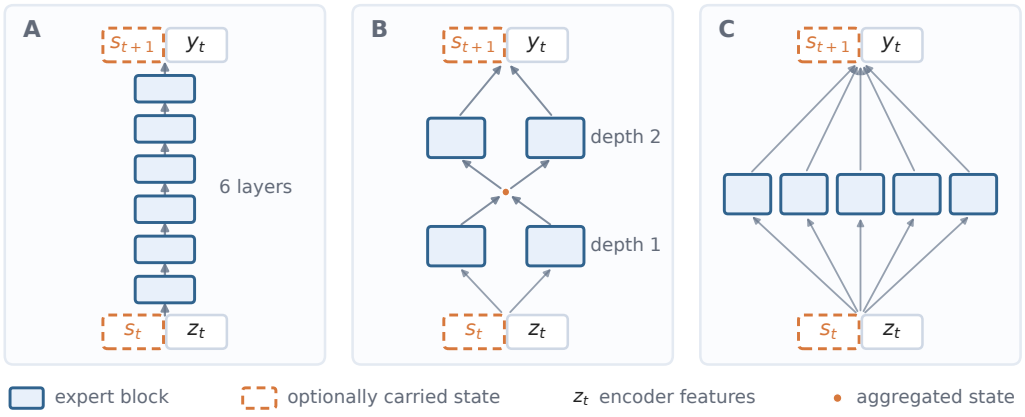


Figure 1: **Three example allocations under the same compute budget.** The three example architectures schematically use the same approximate work budget but allocate it differently. **(A)** applies six expert blocks sequentially. **(B)** uses two depth levels, with two wider experts evaluated in parallel at each level. **(C)** evaluates five experts in parallel within a single depth level. Dashed boxes labeled s_t and s_{t+1} denote latent state that may be carried between external steps in the recurrent variant. Expert-block area represents approximate computational cost, and the total expert-block area is the same across three examples.

2 Why Fewer Layers Matter

The compute-allocation tradeoff between sequential stages S and parallel work P matters also for latency. Under a fixed compute budget, shifting computation from S to P reduces the serial part of

the update, potentially decreasing latency if enough hardware parallelism is available. The actual speedups also depend on implementation details, memory bandwidth, and communication overhead.

This potential reduction in per-update latency is particularly important for general nonlinear RNNs, whose state updates must generally be unrolled sequentially across tokens during language-model pretraining. Because the latency of each update accumulates at every token in the sequence, deeper updates lengthen both the forward unroll and the backward pass through the sequence. If comparable performance can be achieved while reallocating work from sequential stages S to parallel work P, reducing the latency of each update could improve training throughput and make large-scale pretraining of general nonlinear RNNs more practical.

Other sequence architectures address this scalability problem by changing how computation is organized across tokens. Transformers process sequence positions in parallel during training, while structured recurrent models, such as Mamba, constrain their state transitions so that they can be evaluated with parallel scans across the sequence (Vaswani et al., 2017; Gu et al., 2021; Gu & Dao, 2023). These gains in temporal parallelism come with an expressivity trade-off. Under standard precision assumptions and complexity-theoretic conjectures, theory places fixed-depth Transformers and parallelizable linear recurrent models in computational classes believed to be weaker than attainable by general nonlinear RNNs (Merrill et al., 2024; 2026). This trade-off motivates preserving general nonlinear recurrence and studying whether compute can be reallocated from S to P without sacrificing performance, as a potential route to improve training throughput.

There are also pieces of evidence supporting the possibility of parallelization in general. Recent transformer studies suggest that some computations normally implemented as strict depth may be more exchangeable than the architecture implies. Sun et al. (2025) find that middle layers of pretrained transformers share substantial representational structure, and that some tasks tolerate skipping, reordering, or parallelizing layers with limited degradation. Freiberger et al. (2024) train vision transformers to tolerate randomized attention-layer execution order, showing that robustness to arbitrary layer order can be learned, though with an accuracy cost.

3 Preliminaries

We conduct our study in two settings that support updating a latent state over long external sequences: deep reinforcement learning and autoregressive language modeling. In both cases, the model repeatedly receives new input, updates an internal state when recurrence is used, and produces an action or prediction.

Deep Reinforcement Learning. In reinforcement learning, an agent interacts with an environment over timesteps $t = 1, 2, \ldots$ (Sutton et al., 1998; Mnih et al., 2016; Schwarzer et al., 2023). At each step it receives an observation o_t, updates an internal latent state s_t, selects an action $a_t \sim \pi(a_t \mid s_t)$, and receives a reward r_t. The goal is to maximize the expected discounted return, $J(\pi) = \mathbb{E}_\pi \left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t\right]$, where $\gamma \in [0, 1)$ is a discount factor that determines the relative importance of immediate and future rewards. The policy $\pi(a_t \mid s_t)$ maps the current latent state to a distribution over actions, and the expectation is taken over trajectories induced by the policy and environment dynamics. Our experiments use actor–critic agents (Konda & Tsitsiklis, 1999; Mnih et al., 2016).

Recurrent architectures are commonly used to maintain belief-like latent states that summarize trajectory history in partially observable environments and support long-horizon decision making (Hausknecht & Stone, 2015; Heess et al., 2015; Wang et al., 2016; Hefny et al., 2018). In this work, we emphasize a complementary use of recurrent computation in reinforcement learning: recurrence can support reasoning-like latent refinement, not merely memory-like tracking.

Autoregressive Sequence Modeling. Autoregressive sequence modeling aims to predict each token from its preceding context. Given a token sequence $x_{1:T}$, the objective is to model $p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$, or equivalently to predict the next-token distribution at each position (Bengio et al., 2003; Vaswani et al., 2017). This requires representations that accumulate and refine contextual information over long token streams.

Classical recurrent sequence models maintain a latent state across token updates (Hochreiter & Schmidhuber, 1997) and update that state sequentially during training. Modern long-sequence ar-

chitectures often designed to improve training efficiency by enabling parallel scanning computation over sequence positions (Vaswani et al., 2017; Dai et al., 2019; Gu et al., 2021; Gu & Dao, 2023). Our version of the recurrent Transformer is not designed to support parallel scanning; instead, it emphasizes expressive latent-state updates with feedback connections through the carried state. This is intended to allow computation to compose across successive token steps rather than being concentrated within a single deep feed-forward pass.

4 Experimental Framework

We introduce the Parallel Experts framework for studying how compute should be allocated between sequential and parallel computation. It is designed to expose a clean tradeoff between within-step depth and parallel capacity while allowing latent state to be carried across external steps. At each external step, several expert modules read the current input and a shared latent state, compute candidate updates in parallel, and merge their outputs into the shared state. In recurrent models, the final state is carried to the next external step. This construction exposes three allocation axes: within-step depth L, expert count E per depth level, and expert width d.

The depth L counts the number of sequential depth level within one step. The expert count E counts experts evaluated in parallel at each depth level, and d controls per-expert capacity by adjusting expert's hidden dimension (width). If $C(d)$ is the cost of one expert transformation, the approximated work is

$$C \approx L \cdot E \cdot C(d). \tag{1}$$

We use this proxy to construct compute allocation sweeps that are intended to compare where computation should be placed to maximize the performance for recurrent and non-recurrent architectures.

At each environment or sequence step, the model receives an external input x_t. Let $z_t = e(x_t)$ denote the encoded representation of this external input. At step t, the model also receives an incoming shared latent state s_t^0, expert-local activations $h_t^{0,i}$, and architecture-specific memories $m_{t-1}^{\ell,i}$, where $\ell \in \{1, \dots, L\}$ indexes depth levels and $i \in \{1, \dots, E\}$ indexes per depth level parallel experts. Each depth level applies all experts in parallel:

$$\begin{aligned} h_t^{\ell,i},\, m_t^{\ell,i} &= f_{\theta_{\ell,i}}\left(z_t, s_t^{\ell-1}, h_t^{\ell-1,i}, m_{t-1}^{\ell,i}\right), \qquad i = 1, \dots, E, \\ s_t^{\ell} &= \frac{1}{\sqrt{E}} \sum_{i=1}^{E} h_t^{\ell,i}, \qquad \ell = 1, \dots, L. \end{aligned} \tag{2}$$

This normalized sum aggregates expert proposals while keeping the variance approximately the same as the number of contributing experts changes. To maximize the sequential computation available on the carried state, the recurrent variant uses top-down feedback, feeding the final state into the first depth level of the next step (Zilly et al., 2017; Guez et al., 2019; Fan et al., 2020):

$$y_t = g(s_t^L), \qquad h_{t+1}^{0,i} = h_t^{L,i}, \qquad s_{t+1}^0 = s_t^L. \tag{3}$$

Here y_t denotes action logits and a value estimate in reinforcement learning, or next-token logits in language modeling.

More specifically, in the LSTM instantiation $m_t^{\ell,i}$ corresponds to the cell state $c_t^{\ell,i}$, and in the transformer — the key–value (KV) cache. Full pseudocode and architecture-specific instantiations are given in Appendix A and alternative aggregation rules are studied in Appendix D.

The Parallel Experts framework is intended as a probe framework rather than a claim of architectural optimality. We use parallel experts because they expose a clean tradeoff between within-step depth and parallel capacity while optionally allowing a carried latent state. Other MoE mechanisms, including routed MLP experts (Lepikhin et al., 2020), Soft MoE variants (Puigcerver et al., 2023), or alternative recurrent communication schemes, may shift the quantitative frontier, but we expect the qualitative conclusion to stay the same.

5 EXPERIMENTS

Our experiments test different compute allocations. The Parallel Experts framework exposes three ways to spend compute: within-step depth L, per-layer expert count E, or width d of each expert. The central question is whether, once recurrent state is carried across the external steps, the best allocations are still require deep sequential computation within each step.

For our main experiments we evaluate different allocations in two settings: Sokoban, using the `Sokoban-v0` environment from Jumanji (Bonnet et al., 2024), and streamed FineWeb language modeling (Penedo et al., 2024). We report episodic return for reinforcement learning and validation cross-entropy loss for language modeling. Unless stated otherwise, in RL each configuration is run with at least three random seeds. Additional implementation details are reported in Appendix G.

5.1 SOKOBAN

Sokoban is a fully observable puzzle environment in which an agent must push four boxes onto four target locations on a 10×10 grid. Levels are procedurally generated, and mistakes can be irreversible: for example, pushing a box into a corner without a target makes the level unsolvable. Planning is therefore important even though the agent observes the entire board at every step. Prior work showed that DRC agents learn planning-like behavior in Sokoban (Guez et al., 2019), and later analysis found that their internal plans develop progressively through recurrence and causally affect their actions (Bush et al., 2025). These properties make Sokoban a natural setting for studying whether temporal recurrence reduces the within-step depth needed for planning.

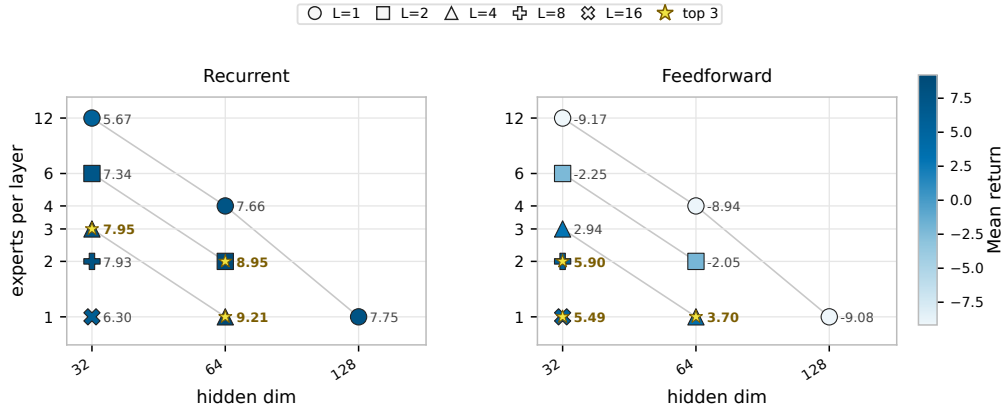


Figure 2: **Compute allocations in Sokoban under the small per-step budget.** We compare recurrent and non-recurrent ConvLSTM agents across different compute allocations, varying within-step depth L, per-layer expert count E, and expert width d under approximately the same compute budget. Higher mean return is better. The best observed recurrent allocation uses four layers and achieves a higher return than the best non-recurrent allocation, which uses eight layers.

We use a DRC-inspired architecture with ConvLSTM experts (Guez et al., 2019), organized according to our Parallel Experts framework. The original DRC setup combines temporal recurrence with depth-recurrence, repeated passes through the recurrent stack on the same observation. To focus on temporal recurrence, we instead make a single pass through the L depth levels per environment step, without additional internal looping. We compare models trained with state carried across environment steps against models trained without carried state, keeping the remaining architecture and training setup the same. For both variants, we vary within-step depth L, expert count E, and expert width d under approximately matched per-step compute budgets. This comparison tests whether temporal recurrence changes how compute is best allocated between within-step depth and parallel capacity.

Fig. 2 shows the allocation sweeps for both variants under the small compute budget. With recurrence, the performance saturates at $L = 4$, while the non-recurrent version saturates at depth 8 with much worse performance. Temporal recurrence changes not only the achieved performance but also the preferred compute allocation, favoring fewer sequential stages and more parallel capacity.

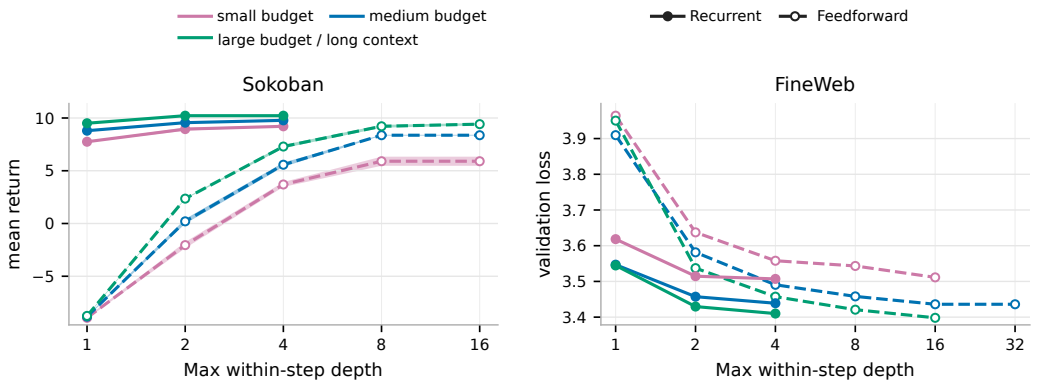


Figure 3: **Within-step depth saturation for recurrent and non-recurrent models.** We report the best result among tested allocations with L or fewer within-step layers, within each compute budget. **Left**: Sokoban mean episodic return with one standard error across three seeds. **Right**: FineWeb validation loss, lower is better. Recurrent models reach their best observed performance with 2–4 layers, while non-recurrent models benefit from greater depth.

We repeat the allocation sweeps at medium and large compute budgets. Fig. 3 summarizes all three budgets, reporting the best observed mean return over expert counts, widths, and depths up to L. Across the tested budgets, recurrent performance saturates by depth 2-4, while non-recurrent models benefit from greater within-step depth, for example, still improving at a within-step depth of 16 under the large budget. At the same time, recurrent model performance is better across all settings, and larger budgets continue to improve both architectures at every tested within-step depth, showing that additional parallel compute remains a useful scaling axis across different configurations. The full medium- and large-budget sweeps are reported in Appendix J.

For a more direct architectural comparison, we also test ConvGLU models, whose experts apply a 3×3 convolution followed by GLU activation and RMSnorm. These models omit the LSTM cell-state machinery retained in the non-recurrent ConvLSTM baseline. Here, the only architectural difference is one extra projection in the recurrent variant, feeding the previous step's final residual stream into the first depth level. We observe the same earlier depth saturation with recurrence under this single architectural intervention (see Appendix K).

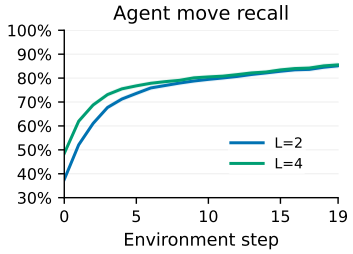


Figure 4: **Agent-move recall.** The four layer model initially has more complete plans, but the two layer model catches up over the next few environment steps. Models performance is almost identical.

To examine how planning develops at different within-step depths, we apply the spatial probes of Bush et al. (2025) to the final residual representations of two large budget recurrent models with $L = 2$ and $L = 4$. The probes predict the direction of the agent's movements in each spatial location. Fig. 4 compares recall across environment steps, with thresholds calibrated to approximately 93% precision. The $L = 4$ model initially has higher recall, but the $L = 2$ model catches up over the next few environment steps, after which both maintain similar recall. Both models also have almost identical performance. This pattern is consistent with the shallower model compensating for less planning within each update by continuing the computation across environment steps, catching up early enough to preserve task performance. Full probing details and additional depth comparisons are provided in Appendix F. We also report preliminary results for fully observable Humanoid in Appendix L.

5.2 Language Modeling

We next test whether the compute-allocation trends observed in Sokoban extend to language modeling. We train recurrent and non-recurrent models for next-token prediction on approximately one billion tokens from FineWeb and compare validation loss under approximately the same per-token compute budgets. We use Transformer blocks within our Parallel Experts framework, with the non-recurrent model reducing to a standard Transformer when $E = 1$. The recurrent variant feeds the

final residual stream into the first layer at the next token step, with additional adjustments for stability and memory efficiency described in Appendix A.

During training, recurrent models process continuous token streams, carrying their state and sliding window KV caches across training sequences and resetting them only at shard boundaries. Non-recurrent models are trained on independent training sequences, allowing parallel computation across token positions. We use fixed KV cache lengths of 128 and 1024 for recurrent models against non-recurrent training sequence lengths of 256 and 2048 tokens, respectively, approximately matching the average attention-context length during training. Both variants use the same total training-token budget and number of tokens per optimizer update. At evaluation, both variants process the validation stream continuously using sliding-window KV caches of the same length (128 or 1024). Further streaming implementation details are provided in Appendix B.

Results. Fig. 5 shows the allocation sweeps under the small compute budget with a 128-token attention window. The recurrent model reaches a validation loss of approximately 3.51 with only two depth levels ($L = 2$), comparable to the best non-recurrent allocation at $L = 16$. As in Sokoban, the preferred allocation shifts toward fewer sequential stages and greater parallel capacity when recurrence is available.

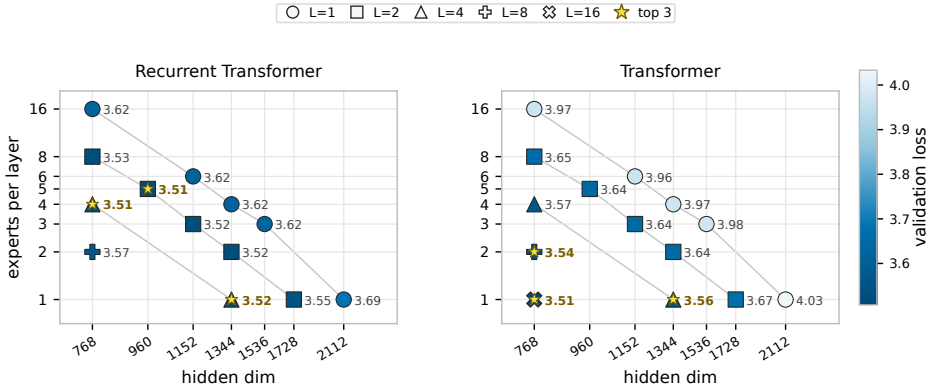


Figure 5: **Compute allocations in FineWeb under the small per-token budget.** We compare recurrent and non-recurrent Transformer allocations trained for next-token prediction on approximately one billion FineWeb tokens. Validation cross-entropy is shown; lower is better. Under approximately matched per-token work, the recurrent model reaches a validation loss of approximately 3.51 with two layers, comparable to the best observed non-recurrent allocation, which uses sixteen layers.

We extend the comparison to a medium compute budget and to the models trained and evaluated with an average 1024-token attention window. Fig. 3 summarizes these settings, reporting the lowest validation loss among allocations with depth at most L. Across the tested settings, recurrent models reach their best observed loss at $L = 2$ or 4, whereas non-recurrent models continue to improve up to $L = 16$ or 32. Although the preferred depth depends on the setting, recurrent models consistently achieve comparable validation loss with substantially less within-step depth.

5.3 What Makes the Earlier Saturation Work?

The early saturation appears to reflect two approximately separable effects. In Fig. 6, increasing per-layer expert count improves both recurrent and non-recurrent models, suggesting that parallel capacity is broadly useful regardless of recurrence. In contrast, the depth-response curve depends strongly on recurrence: non-recurrent performance is much more sensitive to reducing L, whereas recurrent models saturate at substantially shallower within-step depth. Thus, recurrence primarily lowers the cost of reallocating compute away from depth, while experts/width provide a useful parallel place for that compute. Finally, parallel experts must communicate through shared state. In the communication-radius ablation (Appendix E), Full shared-state access performs best while the narrower communication radii lower performance monotonically. Interestingly, Sokoban performance continues to improve beyond the eight-expert regime explored in prior deep RL Soft MoE work (Obando Ceron et al., 2024; Sokar et al., 2025).

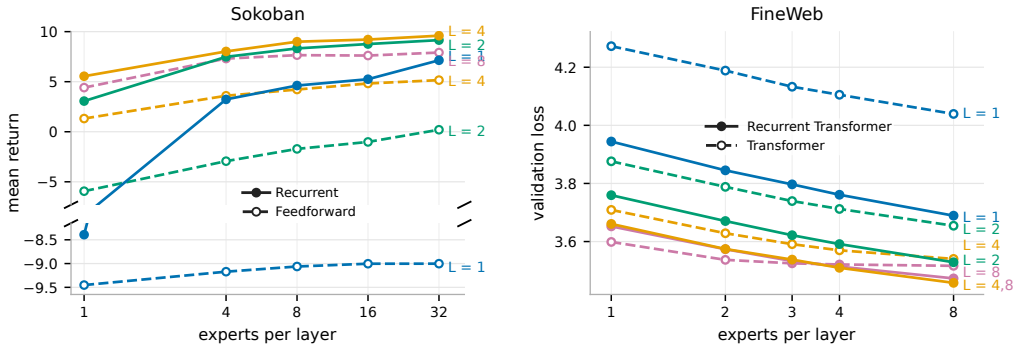


Figure 6: **Scaling within-step depth and experts per layer.** Color denotes within-step depth L. Increasing experts per layer benefits both recurrent and non-recurrent models, while gains from additional within-step depth saturate earlier with recurrence.

6 Related Work

Temporal Recurrence and Model Depth. Several prior works provide evidence that temporal recurrence across external steps, such as token steps in streamed language modeling or environment steps in reinforcement learning, can make shallower recurrent neural networks more competitive with standard feed-forward architectures without temporal recurrence. Feedback Transformer (Fan et al., 2020) expose top-level representations from previous tokens to the current token's computation and show that such feedback makes shallow models with fewer parameters more competitive with standard Transformers that have more layers.

In reinforcement learning, DRC (Guez et al., 2019) carries recurrent state across steps of the Sokoban environment and outperforms deep feed-forward baselines. A closely related classical work, RHN (Zilly et al., 2017), studies how to scale within-step recurrent depth under a fixed parameter budget by trading off depth against width, with depth saturating at approximately 8–9 levels. Importantly, however, compute still grows substantially with depth in the RHN setup, even though the parameter count is held fixed. Similarly, recent recurrent Transformer work (Oncescu et al., 2026) finds favorable depth–width tradeoffs under a fixed parameter budget. In contrast, our work keeps *compute* approximately *fixed*, rather than parameter count, and studies the tradeoff between within-step depth, expert width, and per-layer *expert count* for recurrent and non-recurrent models. In this sense, our work complements and extends prior studies.

Depth Recurrence, Mixture-of-Experts and Compute Allocation. A long line of work studies neural computation as iterative refinement of representations through recurrent/looping transformations on the same input, rather than a single feed-forward pass. Universal Transformers (Dehghani et al., 2019) reformulated Transformer layers as recurrent updates applied across depth, while Adaptive Computation Time (Graves, 2016) and PonderNet (Banino et al., 2021) explored dynamically allocating additional sequential computation to difficult inputs.

A few recent works have also combined depth recurrence with Mixture-of-Experts approaches, motivated by the lack of parameter capacity in these looping architectures; examples include recurrent modular architectures such as Recurrent Independent Mechanisms (RIMs) (Goyal et al., 2019) and recurrent MoE systems such as MoEUT (Csordás et al., 2024). Mixture-of-experts (MoE) architectures scale model capacity through conditional or parallel computation (Jacobs et al., 1991; Jordan & Jacobs, 1994). Modern sparse MoE systems demonstrate that parameter count can scale substantially while keeping per-token computation nearly constant (Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2022; Du et al., 2022). Most prior work integrates experts inside fixed-depth feed-forward architectures, where the dominant scaling axis remains serial layer composition. Other works study improved routing and expert utilization, including differentiable aggregation mechanisms such as Soft MoE (Puigcerver et al., 2023; Obando Ceron et al., 2024; Willi et al., 2024). In our work, we focus mainly on temporal recurrence rather than depth recurrence, and MoE appears in our framework as a way to study a *sequential-to-parallel compute allocation problem*. This perspective is also related to broader work on depth–width tradeoffs and compute-optimal scaling (Lu et al., 2017; Cheng et al., 2017; Hoffmann et al., 2022).

7 Discussion and Limitations

Our results suggest that temporal recurrence does not eliminate sequential computation, but changes where it occurs. Non-recurrent models must complete the computation for each output within the current step, whereas recurrent models preserve and refine intermediate representations as new inputs arrive. In the settings studied here, this shifts the best observed compute allocation from within-step depth and toward parallel capacity. Our Sokoban probing results are consistent with this interpretation: the shallower recurrent agent initially forms less complete plans than the deeper agent, but catches up over subsequent environment steps while achieving comparable task performance.

This tradeoff should depend on task structure. Computation can accumulate across time only when consecutive inputs contribute to a shared problem and the carried state preserves useful intermediate computation. We therefore expect recurrence to reduce the need for within-step depth most clearly in settings with persistent context, progressively evolving representations, or opportunities to refine decisions over multiple steps. The same benefit may not arise when inputs present independent problems, every decision must be solved immediately, or the recurrent state is too limited to retain useful intermediate information.

Our main empirical study covers two domains, a finite set of architectures, and a restricted range of compute budgets, training durations, and task difficulties. The reported allocations are therefore the best among those evaluated, rather than globally compute-optimal configurations. Isolating the effect of recurrence also requires compromises in architectural parity. ConvLSTM contains state machinery that may favor the recurrent variant, while ConvGLU provides a more direct comparison in which recurrence is introduced only through feedback from the previous step's final representation. Similarly, recurrent Transformers require stabilization and memory-management choices that are absent from the standard non-recurrent baseline. Changes in architecture, scale, or architecture-specific tuning may shift the preferred allocation for either model class.

Optimization may also contribute to the earlier depth saturation observed in recurrent models. We train recurrent models using truncated backpropagation through time, which may make credit assignment through deep recurrent updates more difficult (Bengio et al., 1994; Pascanu et al., 2013). Non-recurrent models constructed from similar blocks continue to benefit from greater depth, arguing against a general inability to optimize deep networks, but this does not rule out optimization difficulties specific to recurrence. Finally, our work matched computation rather than realized hardware efficiency. Although reallocating computation from depth to parallel capacity shortens the serial path through each update, actual latency depends on hardware utilization, memory bandwidth, kernel efficiency, and communication overhead. We provide preliminary latency and throughput measurements in Appendix H, but leave a comprehensive systems evaluation to future work.

8 Conclusion

Across Sokoban and streamed FineWeb, temporal recurrence shifts the best observed compute allocation toward substantially fewer layers per step, with comparable or better performance than non-recurrent models under approximately the same compute budget. For example, in the small-budget language modeling FineWeb comparison, a recurrent model with $L = 2$ reaches a validation loss comparable to the best non-recurrent allocation at $L = 16$. Recurrence and parallel capacity play distinct roles in this shift: recurrence reduces the performance cost of using fewer layers, while per-layer expert count and width provide useful ways to spend the reallocated work. Increasing parallel capacity remains useful both with and without recurrence.

Our Sokoban decoding analysis is consistent with computation being redistributed across time, with the shallower model catching up in planning while achieving comparable task performance. Overall, our results support the broader hypothesis that temporal recurrence can shift the best compute allocation toward fewer layers per step in other streaming tasks where useful computation can accumulate across time. This motivates exploring shallower, more parallel updates as a potential route to reducing latency and improving training throughput, making general nonlinear recurrent models more practical at scale.

AI USE STATEMENT

In this work, we used generative AI tools for aiding and polishing paragraph-level writing, discussing and refining the structure and flow of the introduction, finding related work, and editing and writing the code. We also used generative AI tools to assist with early drafts of some sections, which were later substantially rewritten and replaced by the authors. We have not used generative AI tools for research ideation, and the rest of the required disclosure tasks, such as generating synthetic datasets or proving mathematical claims, are not applicable to this work. We have reviewed the AI-assisted work. We reviewed LLM-generated code and read and incorporated suggested related work papers manually. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

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

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence Illing Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries P. Smit, Nathan Grinsztajn, Raphaël Boige, Cemlyn N. Waters, Mohamed A. Mimouni, Ulrich A. Mbou Sob, Ruan de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in JAX. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=C4CxQmp9wc.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

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.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

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

Róbert Csordás, Kazuki Irie, Jürgen Schmidhuber, Christopher Potts, and Christopher D. Manning. Moeut: Mixture-of-experts universal transformers. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 28589–28614. Curran Associates, Inc., 2024. doi: 10.52202/079017-0897. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/321387ba926b8e58d3591c0aeb52ffc2-Paper-Conference.pdf.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 2978–2988, 2019.

Luke Darlow, Ciaran Regan, Sebastian Risi, Jeffrey Seely, and Llion Jones. Continuous thought machines. *Advances in Neural Information Processing Systems*, 38:21548–21594, 2026.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P. Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen S. Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V. Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 5547–5569. PMLR, 2022. URL https://proceedings.mlr.press/v162/du22c.html.

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*, 2020.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Matthias Freiberger, Peter Kun, Anders Sundnes Løvlie, and Sebastian Risi. Layershuffle: Enhancing robustness in vision transformers by randomizing layer execution order. *arXiv preprint arXiv:2407.04513*, 2024.

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, 2026.

Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. Recurrent independent mechanisms. *arXiv preprint arXiv:1909.10893*, 2019.

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

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy P. Lillicrap. An investigation of model-free planning. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2464–2473. PMLR, 2019. URL http://proceedings.mlr.press/v97/guez19a.html.

Matthew J Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *AAAI fall symposia*, volume 45, pp. 141, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

Ahmed Hefny, Zita Marinho, Wen Sun, Siddhartha Srinivasa, and Geoffrey Gordon. Recurrent predictive state policy networks. In *International Conference on Machine Learning*, pp. 1949–1958. PMLR, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022. doi: 10.48550/ARXIV.2203.15556. URL https://doi.org/10.48550/arXiv.2203.15556.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

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

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977. modded-nanogpt: Speedrunning the nanogpt baseline, 2024. URL https://github.com/KellerJordan/modded-nanogpt.

Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

Andrej Karpathy. NanoGPT. https://github.com/karpathy/nanoGPT, 2022.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *Advances in neural information processing systems*, 30, 2017.

William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models. *arXiv preprint arXiv:2404.08819*, 2024.

William Merrill, Hongjian Jiang, Yanhong Li, Anthony Lin, and Ashish Sabharwal. Why are linear rnns more parallelizable? *arXiv preprint arXiv:2603.03612*, 2026.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PmLR, 2016.

Johan Samir Obando Ceron, Ghada Sokar, Timon Willi, Clare Lyle, Jesse Farebrother, Jakob Nicolaus Foerster, Gintare Karolina Dziugaite, Doina Precup, and Pablo Samuel Castro. Mixtures of experts unlock parameter scaling for deep RL. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 38520–38540. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/obando-ceron24b.html.

Costin-Andrei Oncescu, Depen Morwani, Samy Jelassi, Alexandru Meterez, Mujin Kwun, and Sham Kakade. The recurrent transformer: Greater effective depth and efficient decoding. *arXiv preprint arXiv:2604.21215*, 2026.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318. Pmlr, 2013.

Guilherme Penedo, Hynek Kydlícek, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin A. Raffel, Leandro von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/370df50ccfdf8bde18f8f9c2d9151bda-Abstract-Datasets_and_Benchmarks_Track.html.

Joan Puigcerver, Carlos Riquelme Ruiz, Basil Mustafa, and Neil Houlsby. From sparse to soft mixtures of experts. In *The Twelfth International Conference on Learning Representations*, 2023.

Max Schwarzer, Johan Samir Obando Ceron, Aaron Courville, Marc G Bellemare, Rishabh Agarwal, and Pablo Samuel Castro. Bigger, better, faster: Human-level Atari with human-level efficiency. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 30365–30380. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/schwarzer23a.html.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Ghada Sokar, Johan S Obando Ceron, Aaron Courville, Hugo Larochelle, and Pablo Samuel Castro. Don't flatten, tokenize! unlocking the key to softmoe's efficacy in deep rl. In *International Conference on Learning Representations*, volume 2025, pp. 47170–47191, 2025.

Qi Sun, Marc Pickett, Aakash Kumar Nain, and Llion Jones. Transformer layers as painters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25219–25227, 2025.

Richard S Sutton, Andrew G Barto, and Andrew Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

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.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Kevin Wang, Ishaan Javali, Michał Bortkiewicz, Tomasz Trzcinski, and Benjamin Eysenbach. 1000 layer networks for self-supervised rl: Scaling depth can enable new goal-reaching capabilities. *Advances in Neural Information Processing Systems*, 38:157643–157670, 2026.

Timon Willi, Johan Samir Obando Ceron, Jakob Nicolaus Foerster, Gintare Karolina Dziugaite, and Pablo Samuel Castro. Mixture of experts in a mixture of RL settings. In *Reinforcement Learning Conference*, 2024. URL https://openreview.net/forum?id=5FFO6RlOEm.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International conference on machine learning*, pp. 4189–4198. PMLR, 2017.

APPENDIX CONTENTS

A Architectures **15**
A.1 LSTM 15
A.2 Transformer 16
A.3 Recurrent Transformer 16

B FineWeb Streaming Implementation **17**

C Sokoban Solve Rate Results **18**

D Aggregation Ablations **19**

E Communication Radius **20**

F Sokoban Planning Analysis **21**

G Implementation Details and Hyperparameters Used in Experiments **22**
G.1 Compute Resources 23

H Latency and Training Throughput **23**
H.1 Inference Latency 23
H.2 Training Throughput 24

I FineWeb Fixed-Compute Sweeps **24**

J Sokoban Fixed-Compute Sweeps **25**

K ConvGLU Sokoban **26**

L Humanoid Goal-Conditioned RL **28**

A ARCHITECTURES

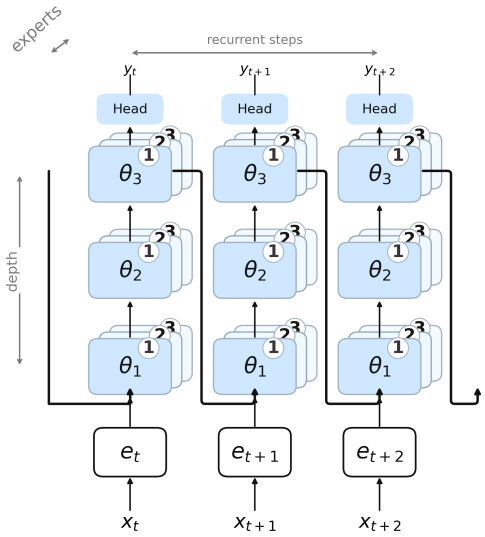


Figure 7: **Architecture.** Each column is one external step. At every step, a fresh input x_t is encoded, within-step experts update the carried state, a head produces output y_t, and the top state is optionally routed into the next external step in case of recurrence.

All architecture variants follow the same pattern: an external input is encoded into z, a shared latent state s is read by several experts in parallel, expert outputs are aggregated, and the updated shared state is carried forward through the external sequence in the recurrent version, as illustrated in Fig. 7. The reinforcement-learning models use LSTM experts, while the FineWeb next-token model uses transformer experts. Figs. 8 and 9 then specialize the expert block for reinforcement learning and FineWeb next-token prediction, and Algorithms 1–2 give the high-level update rules.

A.1 LSTM

Fig. 8 illustrates one depth level with parallel LSTM experts used in reinforcement learning experiments. Each expert receives the encoded observation z, the shared latent state s, and its own local LSTM tensors (h_i, c_i). The experts are evaluated in parallel. At each depth level, their hidden outputs are aggregated into the shared state s, which is passed to the next depth level; the policy and value heads read s after the final depth level. In the non-recurrent LSTM version, all c_i are always zero, and h_i and s inputs for the experts in the first depth level are omitted.

Algorithm 1 Parallel experts LSTM

Encode external input $z \leftarrow e(x_t)$
Receive or initialize shared latent state s, rolling hidden states $h_{1:E}$, and cells $c_{1:E}^{1:L}$
for within-step depth layer $\ell = 1, \ldots, L$ **do**
    for all experts $i = 1, \ldots, E$ in parallel **do**
        $(h_i, c_i^\ell) \leftarrow f_{\theta_i^\ell}(z, s, h_i, c_i^\ell)$
    end for
    $s \leftarrow A(h_1, \ldots, h_E)$
end for
$(\pi_t, V_t) \leftarrow \mathrm{head}(s)$
Return π_t, V_t, updated shared latent state s, and LSTM states $h_{1:E}, c_{1:E}^{1:L}$

Algorithm 1 describes the same computation across L sequential depth levels. The cell c_i^ℓ is local to expert i at depth level ℓ.

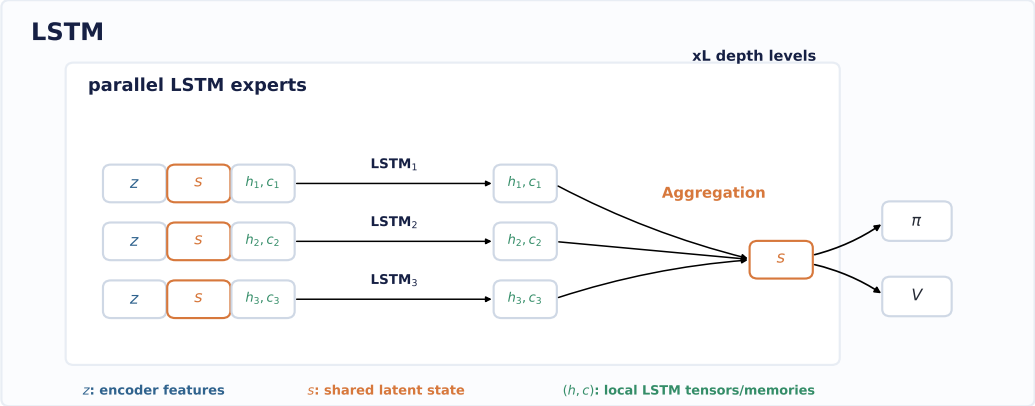


Figure 8: **Parallel experts LSTM.** Each LSTM expert receives encoder features z, the shared latent state s, h_i from the previous depth level or external step and its local recurrent context c_i. At each depth level, expert outputs are aggregated into the shared latent state s, which is passed to the next depth level. The policy and value heads read s after the final depth level.

The Sokoban actor–critic use a spatial recurrence. The observation encoder produces a spatial feature map, and the shared latent state keeps the same board-like spatial structure. The encoder is RMS-normalized over the spatial dimensions. At each environment step, recurrent convolutional LSTM (ConvLSTM) experts read the encoded observation, the shared spatial state, and the incoming hidden state and their previous cell states. The ConvLSTM gates are produced by a 3×3 convolution followed by RMS normalization of the gate preactivations. Expert hidden outputs are aggregated into the shared state, which is carried to the next environment step in case of recurrence or to the next depth level. The Sokoban head aggregates the spatial expert output with attention pooling and global average pooling, applies a dense GLU, and then uses separate linear projections for action logits and the scalar value estimate.

A.2 Transformer

The FineWeb architecture uses standard NanoGPT baseline (Vaswani et al., 2017; Karpathy, 2022; Jordan et al., 2024) components: learned token embeddings, grouped-query attention with RoPE, RMS-normalized queries and keys, a squared-ReLU feed-forward block, final RMSNorm, an LM head, and logit soft-capping. To make a parallel expert version of this non-recurrent Transformer we simply execute E transformer blocks in parallel and then merge the outputs using the aggregation in Eq. 2.

A.3 Recurrent Transformer

To make the recurrent version of it we were needed to introduce two changes in addition to recurrence to stabilize the training, and address memory constrains. We have tried to keep these modifications minimal. First, to stabilize the training each block at all depth levels recieves embeddings of the current token in addition to residual stream. Plus, the output of each expert is normilized with parameter-free RMSNorm. The second addition is that we share the KV cache between experts at each depth level. In our experiments, sharing the KV cache increases validation loss by approximately 0.02, but helps greatly to reduce memory pressure and increases throughput.

In detail, in the parallel expert Recurrent Transformer each expert reads the token embedding z and shared latent state s, forms an expert input, and prepares k_i, v_i, q_i. The per-expert queries remain separate, while the key–value proposals are aggregated across experts into one shared key–value entry. This entry is pushed into a rolling KV cache (K, V) local to that depth level, so all experts at that level attend to the same shared KV cache. The rolling KV cache is a fixed-length FIFO buffer: each new aggregated KV entry is appended, and the oldest entry is discarded once

the buffer is full. The expert attention outputs pass through the transformer feed-forward path, and RMS normalization, producing expert outputs h_i that are aggregated into the next shared latent state. After the final depth level, the shared state is read by the head. The KV caches and shared state are then carried forward by the streamed-document language-modeling pipeline.

Algorithm 2 Parallel experts Recurrent Transformer

Encode token $z \leftarrow e(x_t)$
Receive or initialize shared latent state s and FIFO KV caches $(K^{1:L}, V^{1:L})$
for within-step depth level $\ell = 1, \ldots, L$ **do**
    for all experts $i = 1, \ldots, E$ in parallel **do**
        $(k_i, v_i, q_i) \leftarrow p_{\theta_i^\ell}(z, s)$
    end for
    $k \leftarrow A(k_1, \ldots, k_E)$
    $v \leftarrow A(v_1, \ldots, v_E)$
    $(K^\ell, V^\ell) \leftarrow \text{push_FIFO}\big((K^\ell, V^\ell), (k, v)\big)$
    for all experts $i = 1, \ldots, E$ in parallel **do**
        $h_i \leftarrow g_{\theta_i^\ell}(q_i, K^\ell, V^\ell)$
    end for
    $s \leftarrow A(h_1, \ldots, h_E)$
end for
$\text{logits}_t \leftarrow \text{head}(s)$
Return next-token logits logits_t, shared latent state s, and KV caches $(K^{1:L}, V^{1:L})$

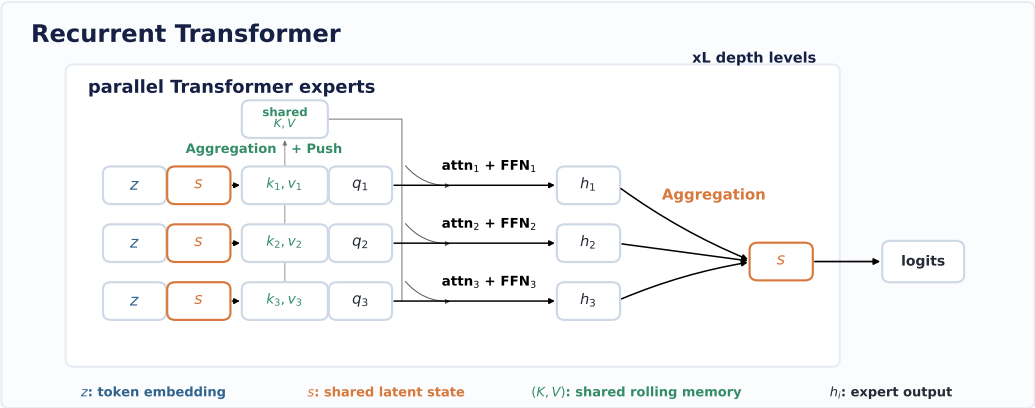


Figure 9: **Parallel experts Recurrent Transformer.** A token embedding z and shared latent state s enter each depth level. Recurrent Transformer experts run in parallel, propose (k_i, v_i, q_i), aggregate and push (k_i, v_i) into a shared rolling (K, V), then each expert attends to that KV cache, and produces output h_i. Expert outputs aggregate into s, which is read by the language-model head.

B FineWeb Streaming Implementation

For a shard with S tokens and global batch size B, the loader partitions the shard into B contiguous streams of length $\lfloor S/B \rfloor$. Batch slot b reads only from stream b. At stream position q, slot b reads a window of length $T + 1$, where T is the training sequence length. The first T tokens are inputs and the next T shifted tokens are labels.

The recurrent state is also organized by batch slot. It is reset at the first window of a shard, carried across later adjacent windows in the same slot, and reset again when a new shard begins.

Algorithm 3 FineWeb shard streaming

Let B be the global batch size and T the training sequence length
for all training shards with S tokens **do**
  Split the shard into B contiguous streams of length $\lfloor S/B \rfloor$
  Initialize recurrent state for all batch slots
  for $q = 0, T, 2T, \ldots$ while all streams have $T + 1$ tokens available **do**
    for all batch slots $b = 1, \ldots, B$ **do**
      $w_b \leftarrow$ tokens from stream b at positions $q : q + T + 1$
      $x_b \leftarrow w_b[0 : T], \quad y_b \leftarrow w_b[1 : T + 1]$
      Reset slot state only if $q = 0$
    end for
    Compute next-token loss and updated recurrent state on $(x_{1:B}, y_{1:B})$
    Update parameters; carry the detached recurrent state to the next window
  end for
end for

C SOKOBAN SOLVE RATE RESULTS

We evaluate Sokoban policies trained on 50M environment steps on the 1,000-level `unfiltered-test` split with ConvLSTM architecture. For each fixed-compute budget and within-step depth $L \in \{1, 2, 4, 8, 16\}$, we select the configuration with the highest mean training return. All selected policies act greedily for at most 120 steps per test level. Table 1 reports the results.

Table 1: **Sokoban ConvLSTM Unfiltered Test solve rates.** Mean and standard deviation over three seeds are reported. State denotes whether recurrent state is carried or reset at every environment step.

Recurrent

Budget	L	E	d	Unfiltered Test (%)
small	1	1	128	86.03 ± 3.71
small	2	2	64	91.93 ± 1.08
small	4	1	64	92.70 ± 1.04
small	8	2	32	86.13 ± 2.40
small	16	1	32	79.00 ± 0.96
medium	1	1	256	93.23 ± 1.06
medium	2	2	128	96.20 ± 0.70
medium	4	1	128	96.67 ± 0.67
medium	8	2	64	93.40 ± 3.84
medium	16	1	64	88.67 ± 1.72
large	1	4	512	96.23 ± 0.91
large	2	8	256	98.47 ± 0.15
large	4	1	512	98.70 ± 0.17
large	8	7	128	95.67 ± 1.44
large	16	13	64	88.57 ± 3.80

Feedforward

Budget	L	E	d	Unfiltered Test (%)
small	1	4	64	3.20 ± 0.00
small	2	2	64	30.50 ± 0.96
small	4	1	64	59.63 ± 3.83
small	8	2	32	74.63 ± 3.91
small	16	1	32	73.23 ± 2.97
medium	1	4	128	3.93 ± 0.12
medium	2	2	128	39.03 ± 0.78
medium	4	1	128	68.93 ± 1.95
medium	8	2	64	86.73 ± 1.34
medium	16	1	64	84.10 ± 4.32
large	1	58	128	4.57 ± 0.50
large	2	2	512	52.93 ± 0.68
large	4	4	256	78.47 ± 0.55
large	8	7	128	92.00 ± 0.26
large	16	13	64	93.00 ± 0.40

For external context, Chung et al. (2023) report Thinker and a reproduction of DRC (Guez et al., 2019) on the same unfiltered Sokoban test set after 50 million real-environment frames. Taufeeque et al. (2024) report DRC(3,3) and ResNet point estimates after two billion environment steps on the same split with the same 120-step episode limit.

Table 2: **Reported Sokoban baselines on Unfiltered Test.** The 50M-frame entries are the mean with one standard deviation reported by Chung et al. (2023); the 2B-step entries are reported by Taufeeque et al. (2024).

Method	Training steps	Unfiltered Test
Thinker	50M	94.5 ± 0.8
DRC(3,3) (Thinker reproduction)	50M	91.3 ± 1.2
DRC(3,3)	2B	99.3
ResNet	2B	97.9

D AGGREGATION ABLATIONS

Aggregation rules. Let h_i denote the output of expert i. For routed variants, α_i denotes the learned softmax routing coefficient for expert i. The four ablations choices are mean pooling, sqrt-normalized sum pooling, softmax-gated weighted averaging, and L2-normalized softmax-gated aggregation. We label these as Mean, Sqrt-sum, Softmax mean, and Softmax L2-sum in Fig. 10:

$$y_g^{\text{mean}} = \frac{1}{m} \sum_{i \in \mathcal{E}_g} h_i, \tag{4}$$

$$y_g^{\sqrt{\text{sum}}} = \frac{1}{\sqrt{m}} \sum_{i \in \mathcal{E}_g} h_i, \tag{5}$$

$$y_g^{\text{rmean}} = \frac{\sum_{i \in \mathcal{E}_g} \alpha_i h_i}{\sum_{i \in \mathcal{E}_g} \alpha_i}, \tag{6}$$

$$y_g^{\text{r}\sqrt{\text{sum}}} = \frac{\sum_{i \in \mathcal{E}_g} \alpha_i h_i}{\sqrt{\sum_{i \in \mathcal{E}_g} \alpha_i^2}}. \tag{7}$$

The sqrt-normalized variants are variance-preserving normalizations: if independent expert outputs have comparable scale, the aggregate scale stays roughly stable as the effective number of contributors changes. The routed mean is the usual softmax-gated weighted average. In the implementation, denominators in the routed formulas are protected with a small ϵ for numerical stability.

Routing coefficients. The coefficients are produced by a scaled dot-product router. For dense experts, the router context is the concatenation of the encoded observation or token representation and the current shared latent state, $c = [x_{\text{enc}}; s]$. For convolutional recurrent experts, both tensors are first spatially average-pooled, then concatenated. The router maps this context to a query and compares it to learned expert keys:

$$q = W_q c + b_q, \qquad k_i = W_k e_i + b_k, \qquad \ell_i = \frac{q^\top k_i}{\sqrt{d}}, \qquad \alpha_i = \text{softmax}(\ell)_i. \tag{8}$$

Here e_i is a learned embedding for expert i, d is the router embedding dimension, and ℓ_i are the router logits.

Default choice. We use Sqrt-sum as the default aggregation rule used throughout the paper, except in experiments that explicitly ablate aggregation. It is parameter-free, keeps the aggregate scale stable as the number of contributing experts changes, and is consistently competitive across domains. For the FineWeb shared-global transformer, we use it for both expert outputs and the merged key-value proposals.

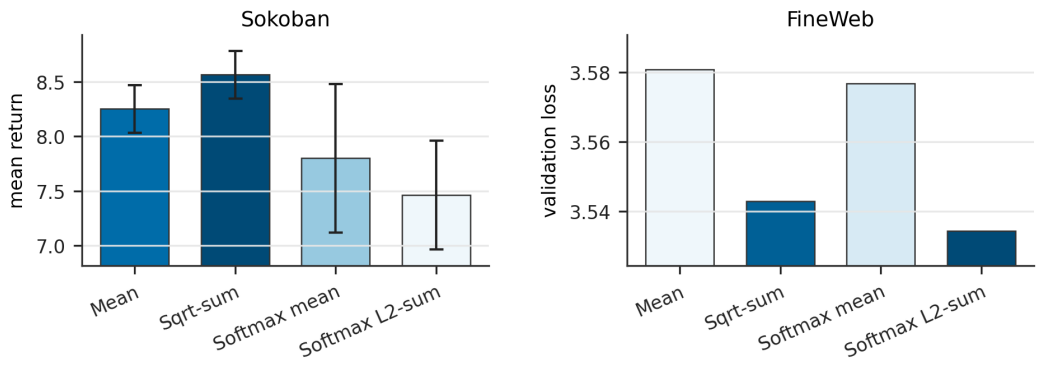


Figure 10: **Aggregation ablations across domains. Left**: Sokoban ConvLSTM with carried state with $d = 128, L = 1, E = 4$; bars show mean episodic return with one standard error across three seeds. **Right**: FineWeb next-token prediction, lower is better.

E Communication Radius

We vary how many other experts each expert can read at each depth level in a Sokoban model with $E = 8$ ConvLSTM experts per layer, width $d = 32$, and within-step depth $L = 2$. Experts are arranged in a ring and read their own state plus those of a chosen number of preceding neighbours. The visible states are summed and divided by the square root of their count, keeping the input width and expert computation fixed.

Fig. 11 shows that full communication achieves the highest mean return. These results indicate that sharing information across experts contributes to the model's performance. We only use full communication for other experiments in the paper.

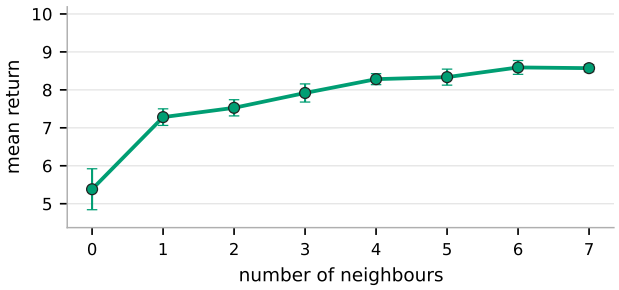


Figure 11: **Communication between experts in Sokoban.** Mean episodic return versus the number of neighbors. Mean returns with one standard error are reported across three seeds.

F SOKOBAN PLANNING ANALYSIS

We follow the spatial probing method of Bush et al. (2025), who decoded iteratively refined plans in DRC agents. We apply their agent-move probes to our models' final residual representations and compare how decoded plans develop across environment steps at different depth allocations.

We extend Fig. 4 to four large-budget recurrent ConvLSTMs with fixed width $d = 256$ and $(L, E) \in (1, 15), (2, 8), (4, 4), (8, 2)$. For each allocation, we select the best of three training seeds by solve rate on 1,000 Unfiltered Test levels, using greedy actions with a 120-step limit.

Following Bush et al. (2025) for each frozen policy, we train convolution 3×3 probes on 3,000 Unfiltered Train trajectories. Each probe reads the final residual representation at each step. At each level location, the target is the direction of the agent in this location in the recorded trajectory, or NEVER if agent does not visit it.

We evaluate decoding on the 906 Test levels solved by all four policies. Recall is the fraction of non-NEVER targets correctly predicted above threshold, and precision is the fraction of emitted directions that are correct. We choose one fixed threshold per model to achieve approximately 93% precision for all prediction over first 20 steps.

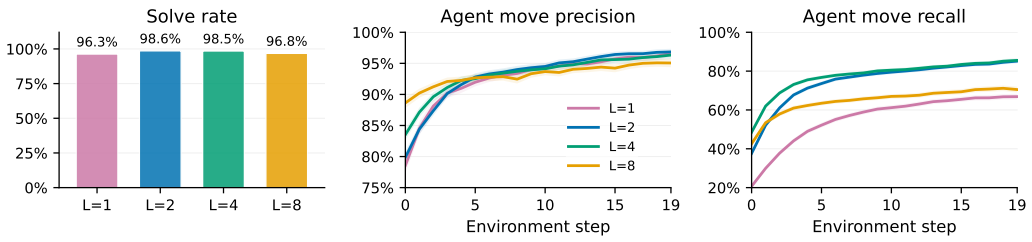


Figure 12: Agent-move decoding comparison across models with different L.

The $L = 4$ model initially has higher recall than the $L = 2$ model, indicating a more complete decoded plan. The $L = 2$ model catches up over the next few environment steps, after which both maintain similar recall. Together with their nearly identical solve rates, this pattern is consistent with the shallower model compensating for less planning within each update by continuing the computation across environment steps, catching up early enough to preserve task performance. A similar pattern appears for the $L = 1$ and $L = 8$ models, which also achieve similar test solve rates: $L = 8$ initially has higher recall, while $L = 1$ narrows the gap over subsequent steps, although it does not fully close it.

Visualization. To make the analysis self-contained, we also include a visualization of how plans develop in the two layer model (Fig. 13). The main DRC model analyzed by Bush et al. (2025) combines within-step looping with temporal recurrence, and our models do not use within-step looping.

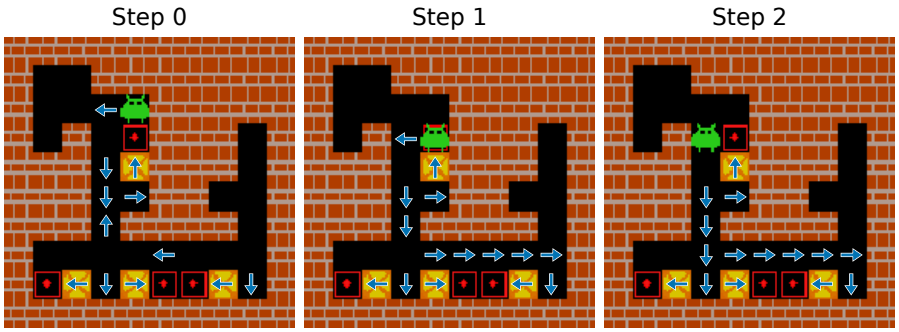


Figure 13: **Agent move plans over the first three environment steps.** The two layer model from Fig. 12 is shown on one Test level.

G Implementation Details and Hyperparameters Used in Experiments

This section reports the training hyperparameters used across the main experiments.

Table 3: Hyperparameters used in reinforcement-learning experiments.

Parameter	**Value**
Sokoban	
Encoder hidden dimension	32
Discount rate γ	0.97
Lambda for general advantage estimation	0.97
Entropy coefficient	0.01
Value function coefficient	1.0
Normalize advantages	No
Unroll length	20
Number of environments	32
Total environment steps	50,000,000
Learning rate	4×10^{-4}
Anneal learning rate	Yes
Optimizer	Adam
Adam epsilon	10^{-6}
Maximum gradient norm for clipping	1.0

Table 4: Hyperparameters used in next-token prediction experiments.

Parameter	Value
FineWeb next-token prediction	
Vocabulary size	50,304
Attention heads	8 query, 4 key-value default; scaled with width proportionally
Token batch size	131,072
Training steps	10,000
Optimizer	Muon core; AdamW embed/readout
Gate parameters	Muon
Optimizer betas	(0.8, 0.95)
Gradient clipping	1.0
Learning rate schedule	Warmup-cosine decay
Peak learning rate	2×10^{-2}
Minimum learning rate	2×10^{-4}
Warmup steps	100
Embedding learning rate	$0.2(d/768)^{-1/2}$
Readout learning rate	$0.004(d/768)^{-1/2}$
Weight decay	0
Cautious weight decay	0.01
Numerical precision	bfloat16
Transformer	
Training sequence length	256 or 2048
Recurrent Transformer	
Shared KV cache length	128 or 1024
Training sequence length	64

G.1 Compute Resources

The reinforcement-learning experiments were run on NVIDIA RTX 8000 and L40S GPUs. Most Sokoban runs use one or two GPUs and take a few hours for small configurations, while the largest fixed-compute and within-step-depth-heavy configurations can take up to roughly one day. The FineWeb next-token prediction experiments were run on NVIDIA H100 GPUs. All FineWeb runs use four GPUs and take several hours for standard configurations, while the largest fixed-compute and within-step-depth-heavy configurations can take up to roughly one day.

H Latency and Training Throughput

These measurements are preliminary. Neither implementation was extensively optimized for inference, so the latency results should be interpreted with caution. Both implementations use JAX; Recurrent Transformer experts at the same depth level are evaluated with `jax.vmap` whenever they can execute in parallel. Measurements use four 80GB NVIDIA H100 GPUs, hidden dimension 768, eight query heads, four key-value heads, and budget $L \cdot E = 16$. For the Recurrent Transformer, L is within-step depth and E is the number of parallel experts per depth level. In latency measurements, B is the global inference batch. In training throughput measurements, B is the global batch size and C is the training sequence length, giving CB tokens per optimizer update.

H.1 Inference Latency

Table 5 reports latency after pre-filling the KV caches of either model to length C.

Table 5: Four-H100 median latency in milliseconds per token.

Model	$C = 128, B = 16$	$C = 2048, B = 16$
Transformer $L = 16, E = 1$	0.565	0.670
Recurrent $L = 1, E = 16$	0.108	0.131
Recurrent $L = 2, E = 8$	0.431	0.554
Recurrent $L = 4, E = 4$	0.552	0.637
Recurrent $L = 8, E = 2$	0.815	0.935
Recurrent $L = 16, E = 1$	1.402	1.577

H.2 Training Throughput

Table 6 measures the complete FineWeb training update, including the normal dataloader, forward and backward passes, and optimizer update.

Table 6: Four-H100 FineWeb training throughput in tokens/s. Each panel has fixed number of tokens per optimizer update. C is the training sequence length and B is the global batch size.

131k effective tokens per optimizer update

Model	$C = 32$ $B = 4{,}096$	$C = 64$ $B = 2{,}048$	$C = 128$ $B = 1{,}024$	$C = 256$ $B = 512$	$C = 1024$ $B = 128$	$C = 2048$ $B = 64$
Transformer $L = 16, E = 1$	967.7k	1,027.4k	1,025.3k	1,054.3k	1,016.9k	968.1k
Recurrent $L = 1, E = 16$	505.4k	445.7k	349.5k	229.4k	51.1k	OOM
Recurrent $L = 2, E = 8$	265.8k	209.5k	145.2k	85.8k	OOM	OOM
Recurrent $L = 4, E = 4$	189.7k	148.7k	99.1k	OOM	OOM	OOM
Recurrent $L = 8, E = 2$	118.3k	88.4k	OOM	OOM	OOM	OOM

262k effective tokens per optimizer update

Model	$C = 32$ $B = 8{,}192$	$C = 64$ $B = 4{,}096$	$C = 128$ $B = 2{,}048$	$C = 256$ $B = 1{,}024$	$C = 1024$ $B = 256$	$C = 2048$ $B = 128$
Transformer $L = 16, E = 1$	886.9k	1,031.5k	1,095.8k	1,132.0k	1,095.8k	1,071.5k
Recurrent $L = 1, E = 16$	556.2k	526.4k	447.7k	313.1k	OOM	OOM
Recurrent $L = 2, E = 8$	306.0k	264.6k	190.2k	OOM	OOM	OOM
Recurrent $L = 4, E = 4$	199.9k	170.8k	OOM	OOM	OOM	OOM
Recurrent $L = 8, E = 2$	OOM	OOM	OOM	OOM	OOM	OOM

I FineWeb Fixed-Compute Sweeps

Fig. 14 show FineWeb fixed-compute allocations at small budget with long context and Fig. 15 shows FineWeb fixed-compute allocations the medium budget, comparing Recurrent Transformer and Transformer architectures.

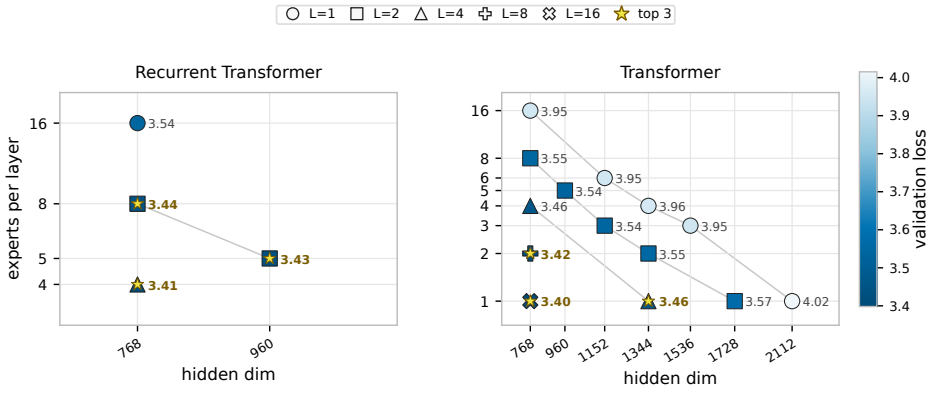


Figure 14: FineWeb small budget allocations with long context. Lower is better.

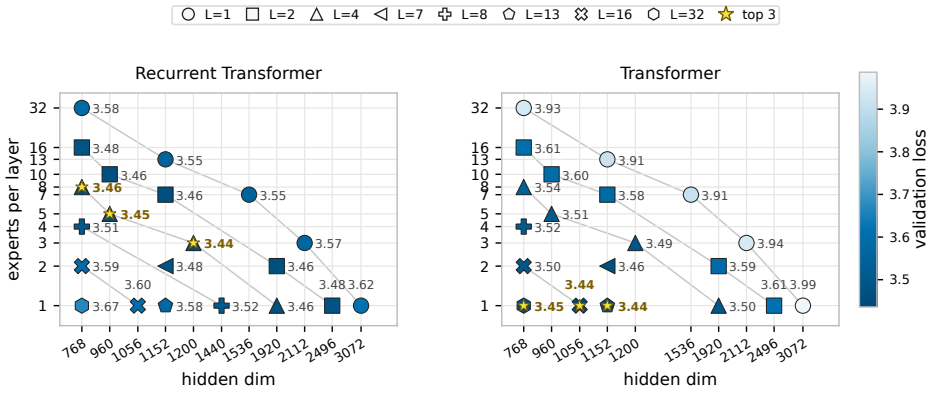


Figure 15: FineWeb medium budget allocations. Lower is better.

J SOKOBAN FIXED-COMPUTE SWEEPS

Fig. 2, Fig. 16 and Fig. 17 show the Sokoban the ConvLSTM allocations at the small, medium, and large budgets under recurrent and feedforward settings. The large-budget snapshot includes only completed training runs.

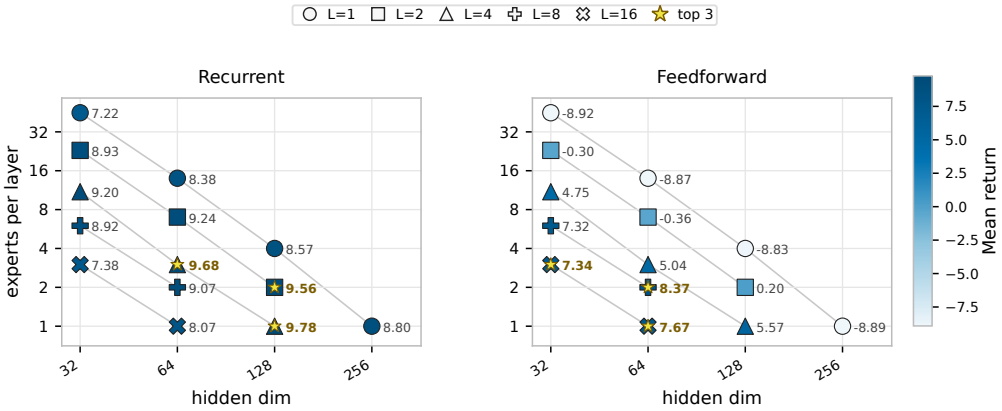


Figure 16: Sokoban medium budget allocations.

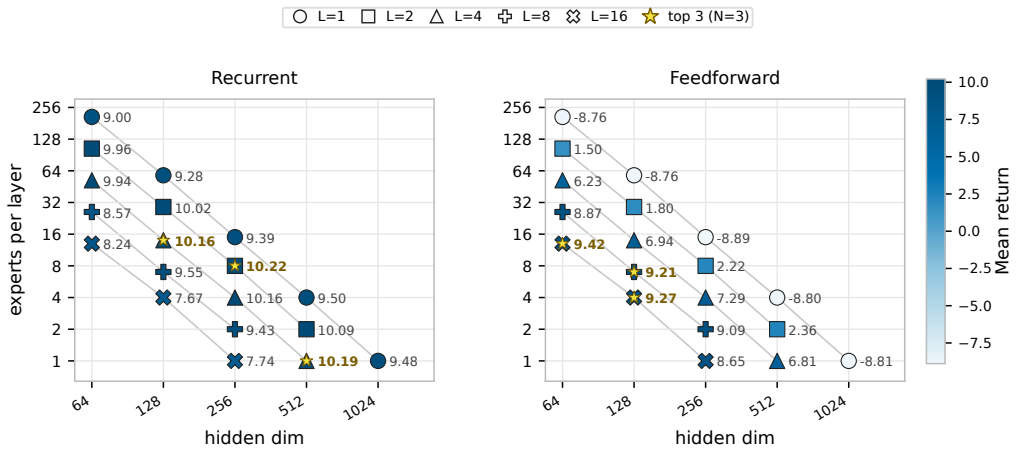


Figure 17: Sokoban large budget allocations.

Table 7: **Sokoban ConvLSTM compute budgets run summaries.** Mean, sample variance (denominator $N-1$), and seed count N of the per-seed final-five episodic returns.

Recurrent

Budget	L	E	d	Mean	Variance	N
Small	1	1	128	7.750	0.0076	3
Small	1	4	64	7.664	0.1200	3
Small	1	12	32	5.670	1.0279	3
Small	2	2	64	8.947	0.0685	3
Small	2	6	32	7.336	0.0454	3
Small	4	1	64	9.212	0.0005	3
Small	4	3	32	7.952	0.4108	3
Small	8	2	32	7.929	0.1292	3
Small	16	1	32	6.297	0.0277	3
Medium	1	1	256	8.795	0.0179	3
Medium	1	4	128	8.566	0.1426	3
Medium	1	14	64	8.382	0.1108	3
Medium	1	45	32	7.224	0.3117	3
Medium	2	2	128	9.558	0.0101	3
Medium	2	7	64	9.241	0.0038	3
Medium	2	23	32	8.928	0.0031	3
Medium	4	1	128	9.778	0.0197	3
Medium	4	3	64	9.677	0.0443	3
Medium	4	11	32	9.200	0.0221	3
Medium	8	2	64	9.067	0.9128	3
Medium	8	6	32	8.919	0.1909	3
Medium	16	1	64	8.068	0.0537	3
Medium	16	3	32	7.378	0.1338	3
Large	1	1	1024	9.479	0.0288	3
Large	1	4	512	9.503	0.0033	3
Large	1	15	256	9.394	0.0132	3
Large	1	58	128	9.278	0.0469	3
Large	1	208	64	9.001	0.0354	3
Large	2	2	512	10.092	0.0085	3
Large	2	8	256	10.219	0.0113	3
Large	2	29	128	10.023	0.0074	3
Large	2	104	64	9.963	0.0267	3
Large	4	1	512	10.189	0.0077	3
Large	4	4	256	10.163	0.0138	3
Large	4	14	128	10.164	0.0074	3
Large	4	52	64	9.944	0.0239	3
Large	8	2	256	9.429	0.4706	3
Large	8	7	128	9.551	0.1403	3
Large	8	26	64	8.575	0.2624	3
Large	16	1	256	7.735	0.1215	3
Large	16	4	128	7.667	0.2837	3
Large	16	13	64	8.241	0.7027	3

Feedforward

Budget	L	E	d	Mean	Variance	N
Small	1	1	128	-9.080	0.0007	3
Small	1	4	64	-8.940	0.0185	3
Small	1	12	32	-9.174	0.0033	3
Small	2	2	64	-2.050	0.2530	3
Small	2	6	32	-2.255	0.2295	3
Small	4	1	64	3.697	0.1589	3
Small	4	3	32	2.938	0.0237	3
Small	8	2	32	5.902	0.5632	3
Small	16	1	32	5.491	0.0821	3
Medium	1	1	256	-8.889	0.0042	3
Medium	1	4	128	-8.829	0.0086	3
Medium	1	14	64	-8.874	0.0006	3
Medium	1	45	32	-8.917	0.0004	3
Medium	2	2	128	0.204	0.1498	3
Medium	2	7	64	-0.357	0.2066	3
Medium	2	23	32	-0.297	0.1262	3
Medium	4	1	128	5.571	0.1257	3
Medium	4	3	64	5.039	0.1118	3
Medium	4	11	32	4.753	0.0918	3
Medium	8	2	64	8.367	0.0050	3
Medium	8	6	32	7.324	0.0180	3
Medium	16	1	64	7.674	0.1781	3
Medium	16	3	32	7.336	0.0311	3
Large	1	1	1024	-8.810	0.0143	3
Large	1	4	512	-8.801	0.0122	3
Large	1	15	256	-8.885	0.0265	3
Large	1	58	128	-8.760	0.0083	3
Large	1	208	64	-8.765	0.0041	3
Large	2	2	512	2.356	0.0064	3
Large	2	8	256	2.216	0.0352	3
Large	2	29	128	1.797	0.0835	3
Large	2	104	64	1.497	0.0736	3
Large	4	1	512	6.807	0.0178	3
Large	4	4	256	7.295	0.0715	3
Large	4	14	128	6.938	0.0739	3
Large	4	52	64	6.226	0.2677	3
Large	8	2	256	9.090	0.0501	3
Large	8	7	128	9.215	0.0544	3
Large	8	26	64	8.875	0.1905	3
Large	16	1	256	8.650	0.1802	3
Large	16	4	128	9.271	0.0152	3
Large	16	13	64	9.417	0.0358	3

K ConvGLU Sokoban

Fig. 18 shows the similar saturation ans scaling effects but for the ConvGLU architecture in Sokoban. Fig. 19 and Fig. 20 show the recurrent and feedforward allocations for the ConvGLU sweeps.

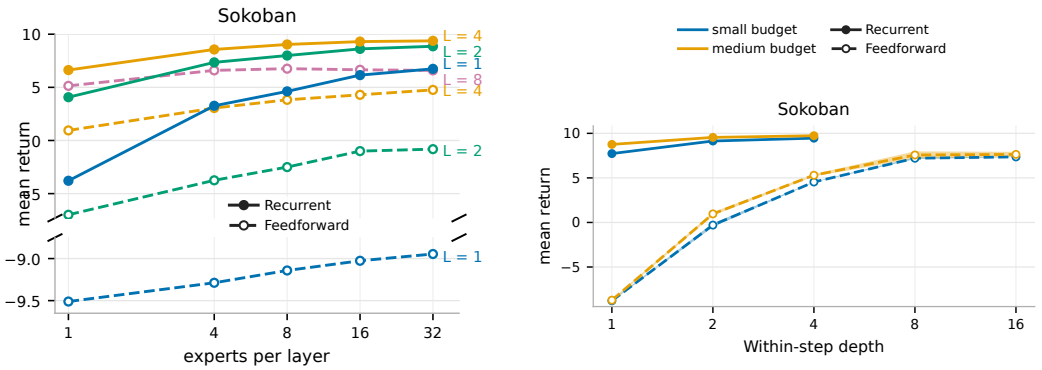


Figure 18: **Sokoban ConvGLU scaling. Left:** Mean return across different numbers of experts per layer at fixed within-step depths. **Right:** Within-step depth saturation at the small and medium budgets. Mean return with one standard error across three seeds is reported.

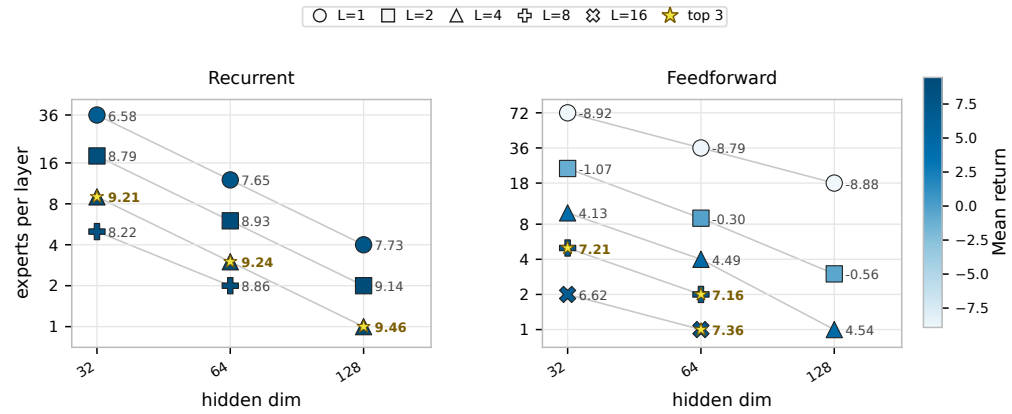


Figure 19: Sokoban small-budget the ConvGLU allocations under recurrent and feedforward settings.

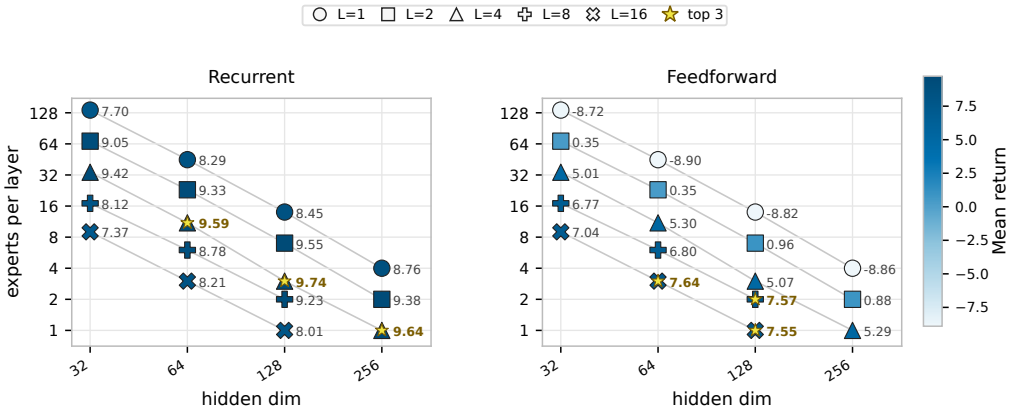


Figure 20: Sokoban medium-budget ConvGLU allocations under recurrent and feedforward settings.

L HUMANOID GOAL-CONDITIONED RL

Recently, Wang et al. (2026) showed that scaling depth in self-supervised contrastive RL (CRL) environments leads to substantially better results. In particular, they use separate actor and critic networks and show that, with critic depth fixed at 32, increasing actor depth from 4 to 32 improves performance from 28 to 636 successful timesteps per episode in their Humanoid experiments.

However, they use feedforward architectures for the actor and critic. In this section, we test whether temporal recurrence can improve the performance of a shallow actor in their Humanoid environment. We hold the critic depth fixed at 32 and compare non-recurrent 32-layer and 4-layer actors with a 4-layer recurrent actor.

Fig. 21 demonstrates that the 4-layer recurrent actor achieves non-trivial performance compared to the 4-layer non-recurrent actor and is not far behind the 32-layer feedforward actor. This observation is consistent with the view presented in this paper that compute can be redistributed through time and that recurrent models favor shallower allocations to reach the same level of performance as deep feedforward models. The 4-layer recurrent model in the figure uses one expert, but we also experimented with 4 and 8 experts (not shown in the figure). More experts helped speed up training, but these models eventually reached a similar level of performance. The results are preliminary.

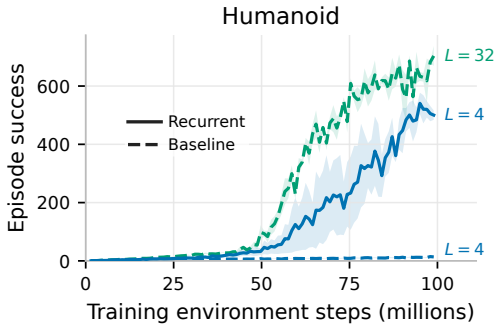


Figure 21: **A shallow recurrent model performs well in CRL Humanoid.** Non-recurrent actors use 4 and 32 layers, and the recurrent actor uses 4 layers with one expert. All models use the same non-recurrent 32-layer critic architecture. Curves show means with one standard error across three seeds. Episode success denotes the mean number of successful timesteps per episode.

Architecture and training modifications. For the non-recurrent critic and actor, the pipeline is identical to the original paper and open-source code (Wang et al., 2026). For the recurrent actor, an additional projection combines the current observation and goal with the previous step's residual stream. For recurrent training, we retain the CRL losses but sample trajectories of length 5 from replay instead of individual observations. Each trajectory uses one shared future achieved goal, and the losses are averaged over its learned steps. This samples five times as many learned timesteps as original training pipeline, so the recurrent and non-recurrent comparisons are not matched for training compute. We also tried this sampling approach with increased compute for the non-recurrent version but found no benefit. Additionally, we found it crucial to clip the recurrent actor's gradients. We use a global norm threshold of 1 for clipping.