INVESTIGATING MEMORY IN RL WITH POPGYM ARCADE

Anonymous authors
Paper under double-blind review

ABSTRACT

How should we analyze memory in deep RL? We introduce mathematical tools for fairly analyzing policies under partial observability and revealing how agents use memory to make decisions. To utilize these tools, we present POPGym Arcade[1], a collection of Atari-inspired, hardware-accelerated, pixel-based environments sharing a single observation and action space. Each environment provides fully and partially observable variants, enabling counterfactual studies on observability. We find that controlled studies are necessary for fair comparisons, and identify a pathology where value functions smear credit over irrelevant history. With this pathology, we demonstrate how out-of-distribution scenarios can contaminate memory, perturbing the policy far into the future, with implications for sim-to-real transfer and offline RL.

1 INTRODUCTION

Real-world decision making often requires reasoning under *partial observability*, where important information is hidden from the agent. This is a challenge for reinforcement learning (RL), which excels in fully observable settings (Silver et al., 2016). To mitigate partial observability, we store and recall sensory information using some form of *memory*. We often evaluate deep memory models by comparing returns across partially observable tasks. However, deep RL experiments are sensitive to model configurations, observation size, task difficulty, and other confounding factors; especially when combined with memory (Mania et al., 2018; Pleines et al., 2025). This sensitivity makes it difficult to isolate the impact of memory on observability, fairly quantify memory performance, or understand failure modes (Jordan et al., 2024).

To address this gap in understanding, we introduce mathematical tools to isolate and study memory and observability, and to interpret how agents use memory. To utilize these tools, we introduce POPGym Arcade (Fig. 1), an Atari-style hardware-accelerated benchmark where every environment has fully observable and partially observable "twins" with identical spaces and dynamics. By combining twins with our memory analysis tools, we perform counterfactual studies on observability. Our findings motivate a secondary study investigating what memory models learn, where we uncover credit attribution failures.

Contributions

- (**C1**) Tools to disentangle, measure, and interpret memory in an RL context.
- (**C2**) POPGym Arcade: a benchmark of paired MDP/POMDP twins for controlled studies.
- (**C3**) Evidence that memory introduces bias and confounds return-based comparisons.
- (**C4**) Discovery of a pathology where memory models incorrectly assign value to irrelevant history.
- (**C5**) Demonstration of recurrent state contamination, where (**C4**) causes irrelevant past to corrupt a policy.

[1]Code available at `https://anonymous.4open.science/r/popgym-arcade-BD90/README.md`

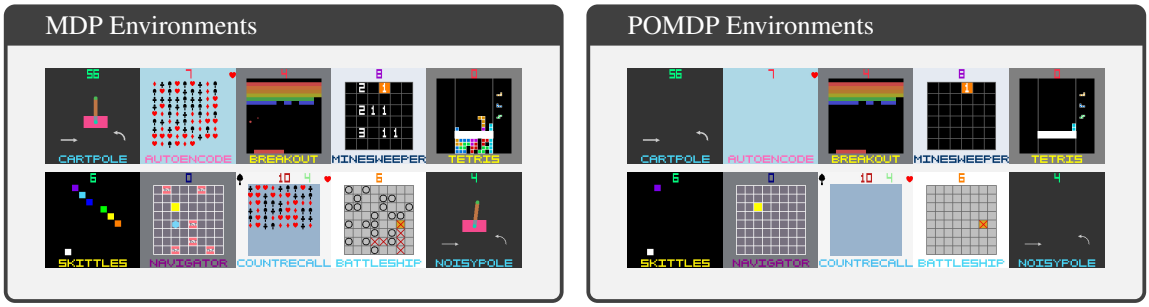


Figure 1: Observations from our environment twins. All environments share a unifying observation and action space, enabling counterfactual studies with observability as the sole independent variable.

2 Preliminaries

Reinforcement Learning A Markov Decision Process (MDP) (Sutton & Barto, 2018) is a tuple $\mathcal{M} = \langle S, A, P, R, \gamma \rangle$, with the set of states S, actions A, and transition function $P : S \times A \mapsto \Delta(S)$, where $\Delta(S)$ is a distribution over S. MDPs also contain a reward function $R : S \times A \mapsto \Delta(\mathbb{R})$ and discount factor $\gamma \in [0, 1)$. At each timestep, we receive a state $s_t \in S$ and select an action $a_t \in A$ from a policy $a_t \sim \pi(\cdot|s_t)$ that takes us to the next state $s_{t+1} \sim P(\cdot|s_t, a_t)$ emitting a reward $r_t \sim R(\cdot|s_t, a_t)$. The Markov property states that s_t contains sufficient information for an optimal policy. In RL, we learn the policy $\pi : S \mapsto \Delta(A)$ that maximizes the expected return $J(\pi, \mathcal{M}) = \mathbb{E}_{\pi,\mathcal{M}}[\sum_{t=0}^{\infty} \gamma^t r_t]$ for $\mathcal{M}$.

POMDPs In Partially Observable Markov Decision Processes (POMDPs), the agent cannot access environment state, instead perceiving the state through noisy or limited sensors. A POMDP $\mathcal{P} = \langle S, A, P, R, \gamma, \Omega, O \rangle$ extends the MDP with set of observations Ω and observation function $O : S \mapsto \Delta(\Omega)$. At each timestep, the agent indirectly accesses the state through a non-Markov observation $o_t \sim O(s_t)$. Fortunately, the trajectory $\mathbf{x}_t = (o_0, a_0, o_1, a_1 \dots a_{t-1}, o_t)$, is itself Markov (Sutton & Barto, 2018). The Reward Memory Length (RML) determines how far back in time we must search to learn the optimal policy (Ni et al., 2024). An RML of k expresses that the reward depends on only the latest k elements of the trajectory $r_{t+1} \perp\!\!\!\perp \mathbf{x}_t, a_t | \mathbf{x}_{t-k:t}, a_t$. Our new RL objective depends on a memory model f and POMDP $\mathcal{P}$: $J(f, \pi, \mathcal{P}) = \mathbb{E}_{\pi,f,\mathcal{P}}[\sum_{t=0}^{\infty} \gamma^t r_t]$.

Memory and Recurrences A memory model f produces a fixed-size latent Markov state $\hat{s} \in \hat{S}$ from the trajectory $\mathbf{x}_t$. This confines the complexities associated with partial observability to f, allowing the policy to operate on a latent Markov state $\hat{s}$. For conciseness, we introduce the joint observation-action space $X = (\Omega, A); x_t = (o_t, a_{t-1})$. A recurrent memory model $f : X \times H \mapsto \hat{S} \times H$ maintains a summary of the trajectory in the recurrent state $h \in H$ and outputs a latent Markov state $\hat{s} \in \hat{S}$ for the policy

$$\hat{s}_t, h_t = f(x_t, h_{t-1}) \qquad a_t \sim \pi(\cdot \mid \hat{s}_t). \tag{1}$$

3 Related Work

POMDP Benchmarks Progress in RL builds upon benchmarks that provide MDPs and POMDPs (Appendix J). Benchmarks include general-purpose tasks (Bellemare et al., 2013; Morad et al., 2023a) and domain specific tasks like navigation (Chevalier-Boisvert et al., 2018; Beattie et al., 2016; Kempka et al., 2016), psychological experiments (Fortunato et al., 2019), and velocity-masked control tasks (Todorov et al.,

2012; Ni et al., 2022), and party games (Pleines et al., 2022). Unfortunately, the cost of training memory models combined with the sample inefficiency of RL is prohibitively expensive, forcing researchers into a tradeoff between simplistic low-dimensional observation spaces and a reduced number of experiments to back their claims. Recent hardware-accelerated benchmarks avoid CPU-GPU copies, resulting in significant training speedups (Hessel et al., 2021; Toledo, 2024), and new high-throughput algorithms like PQN (Gallici et al., 2024). We highlight benchmarks from Matthews et al. (2024a); Pignatelli et al. (2024); Lu et al. (2024); Tao et al. (2025) which offer hardware-accelerated POMDPs.

Controlled Studies of Partial Observability There is ample literature on the theoretical costs and intractability induced by partial observability (Papadimitriou & Tsitsiklis, 1987; Kaelbling et al., 1998; Vlassis et al., 2012; Liu et al., 2022). However, there is less work that empirically quantifies this performance degradation under deep function approximation. Prior work often compares returns between competing memory models (Parisotto et al., 2020; Morad et al., 2023b; Le et al., 2024). With POMDP returns alone, it is not possible to determine whether greater returns are due to more informative latent Markov states or other factors. Paradoxically, literature shows that in certain cases, using memory in MDPs can improve performance, while using memory in POMDPs can reduce performance (Hausknecht & Stone, 2015; Burda et al., 2022; Morad et al., 2023a). For example, a memory model may increase the parameter count of the model, which can improve policy capabilities (Hansen et al., 2023) without improving the Markov state estimate. On the other hand, optimization of memory-endowed policies is notoriously difficult (Mikhaeil et al., 2022), and a suboptimal memory-free policy may reach better local optima in difficult tasks. To isolate the impact of memory, it is necessary to control for all other variables by comparing performance in a partially observable setting against its fully observable counterpart (Jordan et al., 2024). Ni et al. (2022); Morad et al. (2023a); Rajan et al. (2023) provide capabilities for counterfactual studies, but only Ni et al. (2022); Tao et al. (2025) actually perform them. Few works utilize a single shared observation and action space over all tasks, which is necessary to fully control for both parameter count and task difficulty

Investigating Memory in RL Comparing returns indirectly measures the information content of the latent Markov state $\hat{s}$ created by the memory model. Few works directly answer *how* the model constructs the latent Markov state, providing hints but never complete answers. Kapturowski et al. (2019) measure the impact of stale recurrent states. Ni et al. (2024) decouples memory performance from temporal credit assignment, more directly measuring the quality of $\hat{s}$. Finally, Elelimy et al. (2024); Morad et al. (2023b) examine the distribution of recurrent states.

4 Memory Evaluation Tools

The return measures policy performance, but does not disentangle memory capabilities from policy performance, nor does it tell us what the memory model is learning. Below, we describe two tools to isolate and study memory capabilities, and two more tools to understand what memory models learn.

Observability Gap Consider a base MDP $\mathcal{M} = \langle S, A, P, R, \gamma \rangle$. We can induce partial observability by introducing an observation function $o \sim O(s)$, thus creating a corresponding POMDP twin $\mathcal{P} = \langle \mathcal{M}, \Omega, O \rangle$. We are interested in a controlled setting where the observation stream is theoretically sufficient to recover the underlying state, ensuring that the optimal achievable return is identical in both cases.

Our work focuses on the practical difficulty of learning a memory model f, to approximate the latent Markov state. To isolate this challenge, we compare the performance of an agent configuration (π, f) on $\mathcal{M}$ and $\mathcal{P}$. Under $\mathcal{M}$, f must simply propagate states to the policy. The case for $\mathcal{P}$ is more difficult: the agent must infer the Markov state from the trajectory. Any suboptimality in the learned memory model f will result in a performance gap between the two.

Definition 4.1 (Observability Gap)**.** The Observability Gap measures the performance loss of a memory-based agent (f, π) attributable to inferring the state from observations instead of observing it directly

$$\text{Gap}(f, \pi, \mathcal{M}, \mathcal{P}) = J(f, \pi, \mathcal{M}) - J(f, \pi, \mathcal{P}) \quad \text{where} \quad \mathcal{M} = \langle S, A, P, R, \gamma \rangle, \quad \mathcal{P} = \langle \mathcal{M}, \Omega, O \rangle. \tag{2}$$

Memory Bias As stated in Section 3, prior work hints that introducing memory can produce confounding factors. To measure the inherent performance cost of introducing memory, we propose the Memory Bias. This is the observed performance difference between a memory-endowed policy and its memory-free counterpart on a fully observable task. Factors that cause Memory Bias could include parameter count, optimization complexity, implicit regularization, representational capacity, and more.

Definition 4.2. The Memory Bias measures the performance change caused by the memory model that is attributable to the memory model but **not** attributable to partial observability

$$\text{Bias}(f, \pi, \mathcal{M}) = J(f, \pi, \mathcal{M}) - J(\pi, \mathcal{M}). \tag{3}$$

Pixel Visualizations Pixel states and observations enable powerful visualization tools. We implement visualization tools to probe which pixels persist in memory via their impact on the policy. More formally, given a trajectory $\mathbf{x}_n$, we compute latent Markov states $\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n$ via Eq. (1). Then, we backpropagate through memory and policy, taking the absolute value of the gradient of Q values with respect to an observation o_t where $t \leq n$. We provide variants for both Q learning and policy gradient methods

$$\sum_{a_n \in A} \|\nabla_{o_t} Q(\hat{s}_n, a_n)\|_2^2 = \sum_{a_n \in A} \left\| \frac{\partial Q(\hat{s}_n, a_n)}{\partial \hat{s}_n} \frac{\partial \hat{s}_n}{\partial o_t} \right\|_2^2 \quad \int_A \|\nabla_{o_t} \pi(a_n | \hat{s}_n)\|_2^2 \, da_n = \int_A \left\| \frac{\partial \pi(a_n | \hat{s}_n)}{\partial \hat{s}_n} \frac{\partial \hat{s}_n}{\partial o_t} \right\|_2^2 da_n. \tag{4}$$

Eq. (4) measures how much each pixel from a prior observation o_t propagates through memory and contributes to either the Q values or action distribution at time n. We experimented with other norms, but we find the L_2 norm provides the clearest pixel visualizations.

Recall Density Eq. (4) produces qualitative saliency maps, describing pixel-level memory recall over a single trajectory. We are also interested in a more quantitative and consistent measure that elucidates the inner workings of memory in expectation over a distribution of trajectories. For these cases, we introduce the recall density, which we derive below. We provide variants for both Q learning and policy gradient. We start with Eq. (4) and reduce over the observation dimensions using the L_1 norm, such that we receive a scalar time-varying function z

$$z_Q(\mathbf{x}_n, t) = \sum_{a_n \in A} \|\nabla_{o_t} Q(\hat{s}_n, a_n)\|_1 \qquad z_\pi(\mathbf{x}_n, t) = \int_A \|\nabla_{o_t} \pi(a_n | \hat{s}_n)\|_1 \, da_n, \tag{5}$$

where z_Q represents the quantity for Q functions and z_π for policies. Then, we normalize z, representing a scale-invariant density over trajectory $\mathbf{x}_n$

$$\delta_Q(\mathbf{x}_n, t) = \frac{z_Q(\mathbf{x}_n, t)}{\sum_{i=0}^{n} z_Q(\mathbf{x}_n, i)} \qquad \delta_\pi(\mathbf{x}_n, t) = \frac{z_\pi(\mathbf{x}_n, t)}{\sum_{i=0}^{n} z_\pi(\mathbf{x}_n, i)}. \tag{6}$$

Finally, we take the sample mean over m trajectories to approximate the expected density under a policy π and memory model f

$$\frac{1}{m} \sum_{i=1}^{m} \delta_Q(\mathbf{x}_n^{(i)}, t) \approx \mathbb{E}_{\pi, f}[\delta_Q(\mathbf{x}_n, t)] \qquad \frac{1}{m} \sum_{i=1}^{m} \delta_\pi(\mathbf{x}_n^{(i)}, t) \approx \mathbb{E}_{\pi, f}[\delta_\pi(\mathbf{x}_n, t)]. \tag{7}$$

We may compute the resulting expectation over all $0 \leq t \leq n$ to generate a plot summarizing memory recall.

Thus far, n must be equal across all sampled trajectories, while in practice trajectories often differ in length. We use generalized time coordinates to resolve this issue. For each trajectory $\mathbf{x}^{(i)}$ of length n_i, we normalize the timestep $t \in \{0, \dots, n_i\}$ to a value $\tau = t/n_i \in [0, 1]$. We then compute the sample mean of the empirical densities at corresponding normalized time points across m trajectories

$$\frac{1}{m}\sum_{i=1}^{m} \delta_Q(\mathbf{x}_{n_i}^{(i)}, \lfloor \tau \cdot n_i \rfloor) \approx \mathbb{E}_{\pi,f}[\delta_Q(\mathbf{x}, \tau)] \qquad \frac{1}{m}\sum_{i=1}^{m} \delta_\pi(\mathbf{x}_{n_i}^{(i)}, \lfloor \tau \cdot n_i \rfloor) \approx \mathbb{E}_{\pi,f}[\delta_\pi(\mathbf{x}, \tau)]. \tag{8}$$

We may compute the expectation over all $\tau \in [0, 1]$ to generate a plot summarizing memory recall. One should ensure that $\mathbf{x}$ is not terminal, or risk evaluating Q or π at a terminal state.

Definition 4.3 (Recall Density)**.** The Recall Density measures the expected relative influence of a past observation at the current timestep. For trajectories generated by policy π and memory model f, it quantifies how much an observation at normalized time $\tau \in [0, 1]$ impacts either:

(4.3.1) $\mathbb{E}_{\pi,f}[\delta_Q(\mathbf{x}, \tau)]$: The Q value at trajectory end, via Q gradient magnitudes

(4.3.2) $\mathbb{E}_{\pi,f}[\delta_\pi(\mathbf{x}, \tau)]$: The action distribution at trajectory end, via policy gradient magnitudes

5 POPGym Arcade

We cannot utilize our memory analysis tools with most prior benchmarks as the Observability Gap requires MDP/POMDP twins and pixel visualizations require both pixel states and observations. To this end, we propose POPGym Arcade (Table 1 and Appendices H and I). POPGym Arcade consists of ten base environments each with three difficulty levels. All environments are inspired by existing human-playable card games, board games, and video games, with each environment testing different aspects of memory. Unlike Atari, our tasks utilize (1) highly stochastic transition functions (2) hardware acceleration (3) known RML (4) standardized returns to enable easy comparisons.

Base Envs	Total Envs	State/Observation Size	Action Space	Return Bounds
10	120	S: $128 \times 128 \times 3$ L: $256 \times 256 \times 3$	$\uparrow, \downarrow, \leftarrow, \rightarrow, \text{Fire}$	[-1, 1]

Base Env	Notes & POMDP Utility	RML
Autoencode	Push and pop to latent stack, NC^1 circuit complexity	$O(n)$
CartPole	Classic deterministic control task, useful for debugging	$O(n)$
CountRecall	Increment and query latent counters, TC^0 circuit complexity	$O(n)$
BattleShip	Large configuration space and highly-stochastic transition function	$O(n)$
Breakout	Disappearing ball, constant RML, and long episodes	$O(k)$
MineSweeper	Difficult games rules, requires in-context exploration	$O(n)$
Navigator	Navigation and path planning, requires latent map representation	$O(n)$
NoisyPole	State estimation under perturbations, surrogate for real-world control	$O(n)$
Skittles	Obstacle avoidance with flickering obstacles, requires path replanning	$O(k)$
Tetris	Hard spatial task; human competitions for MDP & POMDP	$O(n)$

Table 1: A summary of our environments. We provide both small (S) and large (L) state/observation spaces for each task. We describe RML over episode length n or a constant length k. Frame stacking or fixed-window attention are sufficient to solve $O(k)$ RML POMDPs but not $O(n)$ POMDPs.

Environment Twins Our twins introduce two notions of state: a low dimensional hidden Markov state $\tilde{s} \in \tilde{S}$ and a pixel Markov state $s \in S$. For example, in MDP MineSweeper, $\tilde{s}$ contains the location of all mines, while s contains pixels representing numbered tiles inferring mine locations. Both states are Markov, but deterministic transitions in $\tilde{S}$ become stochastic in S. While $\tilde{S}$ varies across tasks, S is identical for all tasks. By decoupling $\tilde{s}$ from s, we can introduce POMDP variants with an observation function $O : \tilde{S} \mapsto \Delta(\Omega)$. Consequently, twins share identical underlying transition dynamics $\tilde{S} \times A \mapsto \tilde{S}$. All tasks share a singular state/observation space $S = \Omega$ and action space A. As a result, we can reuse a single model across all tasks and task configurations. We can even change from POMDP to MDP mid-episode, or train a single policy on both MDP and POMDP at the same time. The motivation behind this task structure is to isolate and study memory and observability.

6 Results

We measure and analyze memory with our proposed tools, using the PQN algorithm (Gallici et al., 2024) across five random seeds. We describe the experimental setup in Appendices L, M and O and memory models in Appendix K.

6.1 Disentangling Returns

What portion of the return can we attribute to memory models closing the observability gap, and what portion can we attribute to other factors? In Fig. 2, we plot the return, Observability Gap, and Memory Bias (Definitions 4.1 and 4.2) aggregated over all tasks and difficulty levels.

We find that Memory Bias varies between models and can spoil return-only comparisons. For most models, the Gap and Bias have similar scales, demonstrating the weakness of return-only comparisons and the utility of disentangled metrics. Take the MinGRU and GRU, for example. The relative effect of Bias $0.07 - 0.02 = 0.05$ impacts the return as strongly as the Observability Gap $0.16 - 0.11 = 0.05$. Purely negative biases across models points to a memory-induced performance penalty, rather than benefit. Although memory models increase parameter count and can approximate a wider class of functions than feedforward networks, such effects are not strong enough to overcome other negative factors. Looking deeper, we note a correlation between the return, Gap, and negative Bias, suggesting a latent factor affecting all three.

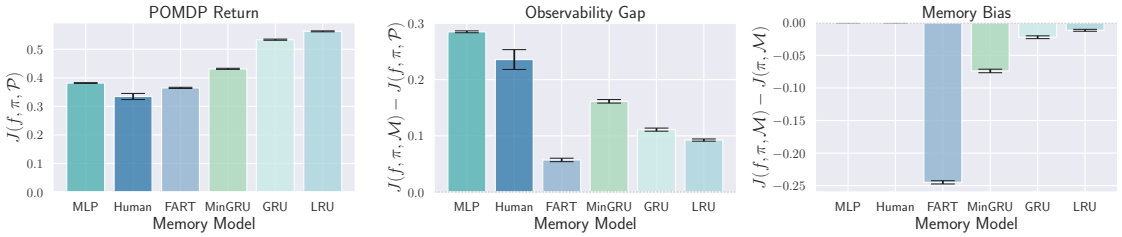


Figure 2: We disentangle the return using our memory analysis tools. We plot the POMDP returns $\in [0, 1]$, the Observability Gap, and Memory Bias. We aggregate scores over all environments and difficulty configurations. Whiskers represent the 95% confidence interval over five seeds.

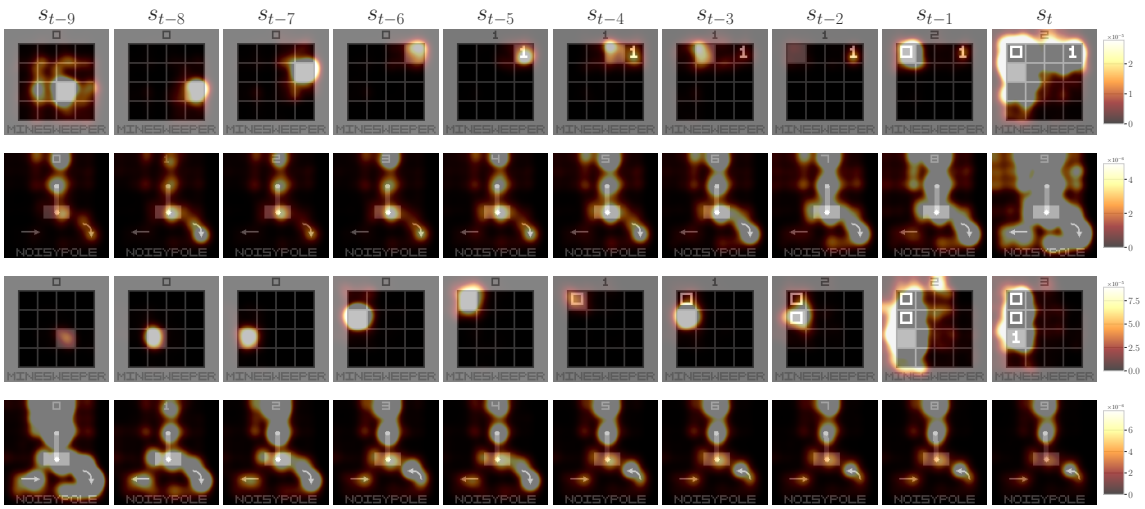


Figure 3: How do trained agents use memory? We plot pixelwise memory gradients Eq. (4) for the LRU (top rows) and GRU (bottom rows), denoting how much each pixel contributes to future value estimate $V(s_t)$ through memory. In these MDPs, $V_*(s_t)$ is independent of $s_{t-k} \dots s_{t-1}$, yet memory incorrectly smears value credit over uninformative past states, even with a residual connection bypassing the memory model. Smeared value attribution suggests value estimators may not generalize to new trajectories.

6.2 The Value Smearing Pathology

Next, we investigate how our trained policies use memory with our gradient visualization tools from Eq. (4). We train policies long beyond convergence (Appendix F) and perform qualitative pixel-level gradient visualizations on both MDPs and POMDPs (Fig. 3 and Appendix C). We highlight the MDP results because MDPs have a known ground-truth credit distribution: the return depends only on the current state/observation.

The resulting gradient distributions appear "smeared" over all prior states, even those containing no useful information (e.g., empty board denoting only past actions). We call this phenomena "value smearing". Value smearing demonstrates that the policies struggle to decorrelate past actions from the forward-looking return, and may signal that the memory model and Q function may have overfit to the trajectory distribution induced by the current policy.

One may wonder if these results are cherry-picked. Fortunately, we can use the Recall Density (Definition 4.3) to measure memory access in expectation. In Fig. 4, we average results across task, memory model, and seed, finding evidence of value smearing across all models and tasks. States from the first two-thirds of an episode ($\tau < 0.66$) impart a significant impact on Q values across all MDPs. See Appendix B for a per-model per-task breakdown.

6.3 Recurrent State Contamination

Naturally, we ponder what practical consequences arise from value smearing. Perhaps it induces robustness to Out of Distribution (OOD) scenarios. By distributing credit diffusely, a single OOD observation may impart little impact on the policy. To test this hypothesis, we first collect trajectories following learned policies. Then, we perturb these collected trajectories similar to Stone et al. (2021) and analyze how the relative (mean-centered) Q values $A(s,a) = Q(s,a) - |A|^{-1} \sum_{a' \in A} Q(s,a')$ and policy change. We measure the

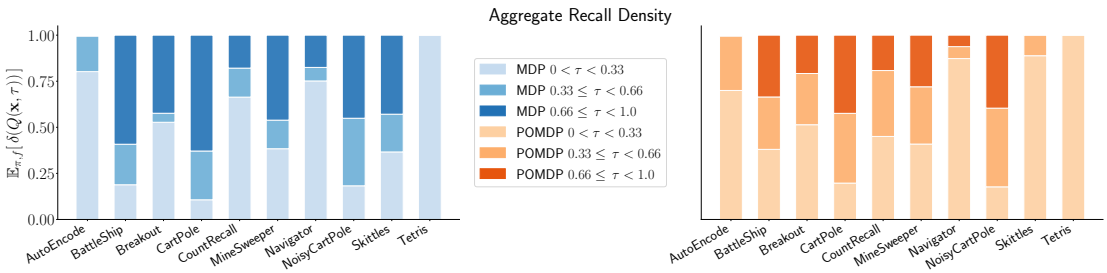


Figure 4: How does the past affect future value predictions? We bin trajectories into thirds, and plot the contribution of each bin on the final Q value. We estimate recall density $\mathbb{E}_{\pi,f}[\delta_Q(\mathbf{x},\tau)]$ for the start, middle, and end segments $\tau \in [0, 0.33), [0.33, 0.66), [0.66, 1.0)$ of a trajectory, aggregating across models and seeds. All density for MDPs should be in $0.66 \leq \tau < 1.0$, given the Markov property. Instead, we see credit diffusely distributed across trajectories for all models and tasks, demonstrating the value smearing pathology.

impact of (1) adding noise to a single frame in each trajectory (Fig. 5a and Appendix E) and (2) shuffling the first few observations in each trajectory (Fig. 5b). Observation noise is a common problem in POMDPs, but it is unclear whether the RNN or CNN is responsible for any robustness or sensitivity. By shuffling the beginning of the trajectory, any observed effects will be purely caused by memory.

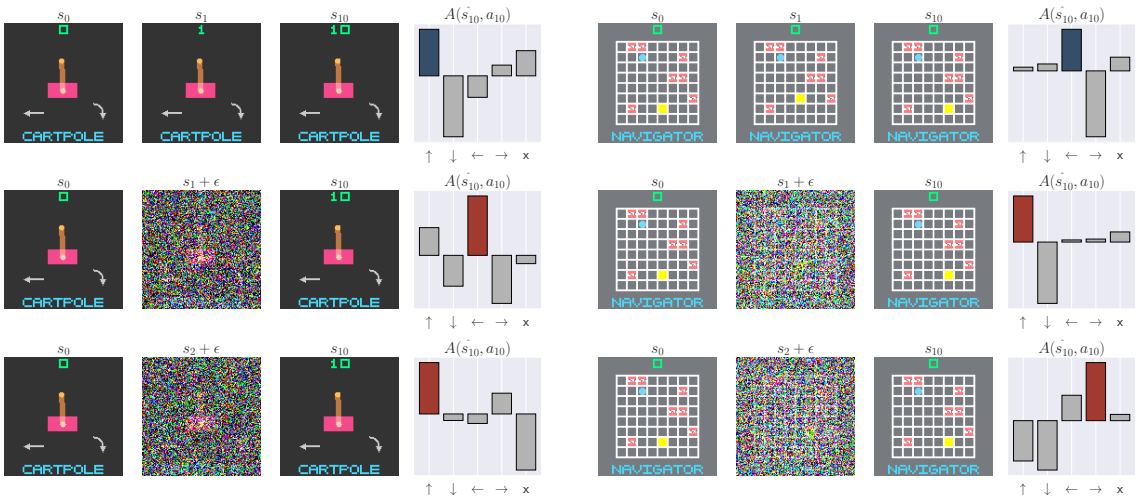
We find that recurrent value functions are not robust and they do not smooth over OOD observations. Even a single OOD observation can be disastrous to the policy, "contaminating" the recurrent state, perturbing the policy far into the future. We see similar results even when we permute the trajectory, removing possible CNN confounders. Our results demonstrate large sensitivity to both OOD observations and trajectories.

7 Limitations and Future Work

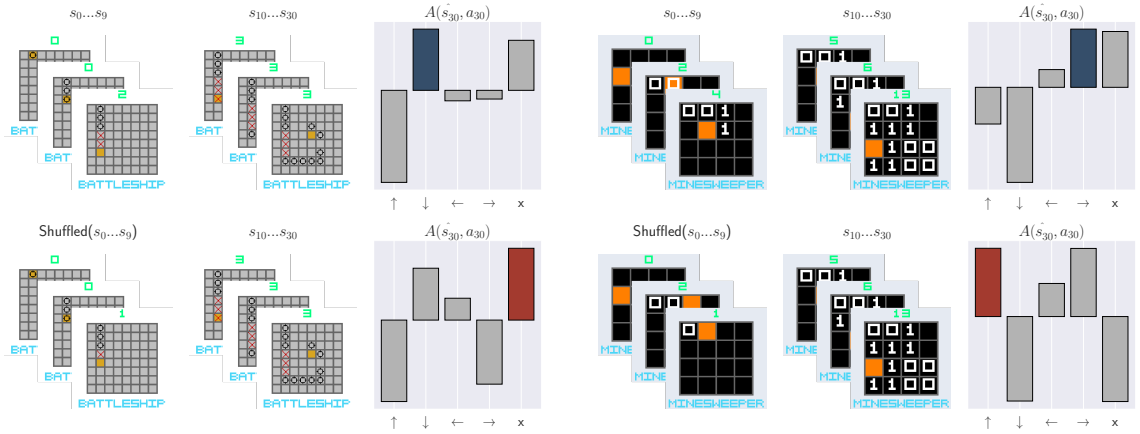
Understanding Memory Bias Correlations between all three POMDP metrics across memory models suggest that a latent factor may be responsible for negative memory bias. Value smearing and brittle policies hints that optimization difficulty could be one possibility. However, proving causation is difficult and requires future investigation. To avoid confounders, future studies should avoid return-only comparisons and utilize the Gap and Bias metrics to measure model capabilities.

Recurrent State Contamination Our contamination experiments are proofs of concept, not a rigorous analysis. That said, these findings have implications for offline RL, imitation learning, or sim-to-real scenarios, where agents may encounter out-of-distribution observations or trajectories after training. We focused on recurrent models primarily due to hardware constraints, but a transformer policy will discard out-of-distribution observations when they exceed the context window, potentially reducing contamination effects. Future work could investigate if such phenomena still occur with fixed-context, non-recurrent memory models.

Model-Based RL Our study primarily uses PQN, a simple model-free Q learning algorithm, to demonstrate our analysis tools. This was a deliberate choice to isolate pathologies in value estimation from confounding factors in more complex agents. Model-based RL methods learn transition and reward functions using different objectives (Hafner et al., 2025; Samsami et al., 2024), and may not susceptible to the shortcomings we observe in model-free RL. Future studies are necessary to understand learning dynamics for model-based RL.



(a) We add noise ϵ into one observation and examine how this perturbation propagates into current values and actions.



(b) We shuffle the beginning of each trajectory to remove any CNN confounders. Even in-distribution observations can contaminate the recurrent state when the trajectory is OOD.

Figure 5: OOD scenarios contaminate the recurrent state and perturb the policy. The rightmost columns plot the relative Q values (A), and we color the policy action. We see that not only do the future values change significantly, but so does the policy action. We expect to experience OOD trajectories in offline RL and sim-to-real scenarios.

8 Conclusion

We introduced POPGym Arcade and a methodology to dissect memory in RL, revealing that return-based comparisons can be misleading. We proposed alternative metrics that tend to be more fair when making memory model comparisons. Then, we identified a value smearing pathology, where value is incorrectly attributed to irrelevant history. We demonstrated that this credit assignment failure makes memory-endowed policies sensitive to OOD scenarios, corrupting decisions far into the future.

Ethics Statement

We collected human baseline results by recruiting five participants to play our benchmark tasks. Each participant was provided with the docstring associated with each game and interacted using the arrow keys and spacebar. Scores were recorded and analyzed for comparison with algorithmic baselines. All participants gave informed consent, no personally identifiable information was collected, and participation posed minimal risk, as the tasks involved only standard computer-based gameplay. Compensation was provided at fair rates. The study adhered to institutional and community ethical guidelines for research involving human subjects.

Reproducibility Statement

We provide memory analysis scripts and full benchmark code in the anonymized supplementary materials. We describe memory evaluation tools and POPGym Arcade in main text, while Appendix H gives detailed environment descriptions, Appendix M reports all experiment hyperparameters, and Appendix N outlines our hyperparameter selection methodology. Results are averaged across five random seeds, with additional analyses in Appendices A to C and E to G. Human baseline procedures are documented in Appendix O, and compute resources are detailed in Appendix P. Together, these materials allow independent researchers to reproduce and extend our findings.

REFERENCES

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. DeepMind Lab. Technical Report arXiv:1612.03801, arXiv, December 2016. URL http://arxiv.org/abs/1612.03801. arXiv:1612.03801 [cs] type: article.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL https://www.jair.org/index.php/jair/article/view/10819.

Guy E Blelloch. Prefix Sums and Their Applications. Technical report, School of Computer Science, Carnegie Mellon University, November 1990.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence I. Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries P. Smit, Nathan Grinsztajn, Raphael 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, 2024. URL https://arxiv.org/abs/2306.09884. _eprint: 2306.09884.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, 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.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by Random Network Distillation. February 2022. URL https://openreview.net/forum?id=H1lJJnR5Ym.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision Transformer: Reinforcement Learning Via Sequence Modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic Gridworld Environment for OpenAI Gym, 2018. URL https://github.com/maximecb/gym-minigrid. Publication Title: GitHub repository.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

Esraa Elelimy, Adam White, Michael Bowling, and Martha White. Real-Time Recurrent Learning using Trace Units in Reinforcement Learning. November 2024. URL https://openreview.net/forum?id=4UvMOnZMam&noteId=SpwUrmTQx1.

Leo Feng, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadeghi. Were RNNs All We Needed? October 2024. URL https://openreview.net/forum?id=GrmFFxGnOR.

Meire Fortunato, Melissa Tan, Ryan Faulkner, Steven Hansen, Adrià Puigdomènech Badia, Gavin Buttimore, Charles Deck, Joel Z Leibo, and Charles Blundell. Generalization of Reinforcement Learners with Working and Episodic Memory. In *Advances in Neural Information Processing Systems*, pp. 12448–12457, 2019.

Matteo Gallici, Mattie Fellows, Benjamin Ellis, Bartomeu Pou, Ivan Masmitja, Jakob Nicolaus Foerster, and Mario Martin. Simplifying Deep Temporal Difference Learning, October 2024. URL http://arxiv.org/abs/2407.04811. arXiv:2407.04811 [cs].

Albert Gu, Karan Goel, and Christopher Re. Efficiently Modeling Long Sequences with Structured State Spaces. March 2022. URL https://openreview.net/forum?id=uYLFoz1vlAC.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640(8059):647–653, April 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-08744-2. URL https://www.nature.com/articles/s41586-025-08744-2. Publisher: Nature Publishing Group.

Nicklas Hansen, Hao Su, and Xiaolong Wang. TD-MPC2: Scalable, Robust World Models for Continuous Control. October 2023. URL https://openreview.net/forum?id=Oxh5CstDJU.

Matthew Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. In *2015 AAAI Fall Symposium Series*, September 2015. URL https://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11673.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable Reinforcement Learning, April 2021. URL http://arxiv.org/abs/2104.06272. arXiv:2104.06272 [cs].

Ralf Hinze. An Algebra of Scans. In *Mathematics of Program Construction: 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004. Proceedings 7*, pp. 186–210. Springer, 2004.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Scott M. Jordan, Adam White, Bruno Castro Da Silva, Martha White, and Philip S. Thomas. Position: benchmarking is limited in reinforcement learning research. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *ICML'24*, pp. 22551–22569, Vienna, Austria, July 2024. JMLR.org.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2):99–134, 1998. ISSN 00043702. doi: 10.1016/s0004-3702(98)00023-x.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent Experience Replay in Distributed Reinforcement Learning. In *International conference on learning representations*, 2019.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In *Proceedings of the 37th International Conference on Machine Learning*, pp. 5156–5165. PMLR, November 2020. URL https://proceedings.mlr.press/v119/katharopoulos20a.html. ISSN: 2640-3498.

Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. In *IEEE Conference on Computational Intelligence and Games*, pp. 341–348, Santorini, Greece, September 2016. IEEE. URL http://arxiv.org/abs/1605.02097.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 45716–45743, 2023.

Robert Tjarko Lange. gymnax: A JAX-based Reinforcement Learning Environment Library, 2022. URL http://github.com/RobertTLange/gymnax.

Hung Le, Dung Nguyen, Kien Do, Sunil Gupta, and Svetha Venkatesh. Stable Hadamard Memory: Revitalizing Memory-Augmented Agents for Reinforcement Learning. October 2024. URL https://openreview.net/forum?id=We5z3UEnUY.

Qinghua Liu, Alan Chung, Csaba Szepesvari, and Chi Jin. When Is Partially Observable Reinforcement Learning Not Scary? In *Proceedings of Thirty Fifth Conference on Learning Theory*, pp. 5175–5220. PMLR, June 2022. URL https://proceedings.mlr.press/v178/liu22f.html. ISSN: 2640-3498.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered Policy Optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, December 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/688c7a82e31653e7c256c6c29fd3b438-Abstract-Conference.html.

Chris Lu, Yannick Schroecker, Albert Gu, Emilio Parisotto, Jakob Foerster, Satinder Singh, and Feryal Behbahani. Structured State Space Models for in-Context Reinforcement Learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://papers.nips.cc/paper_files/paper/2018/hash/7634ea65a4e6d9041cfd3f7de18e334a-Abstract.html.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Thomas Jackson, Samuel Coward, and Jakob Nicolaus Foerster. Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning. June 2024a. URL https://openreview.net/forum?id=hg4wXlrQCV.

Michael Matthews, Michael Beukman, Chris Lu, and Jakob Foerster. Kinetix: Investigating the Training of General Agents through Open-Ended Physics-Based Control Tasks, October 2024b. URL http://arxiv.org/abs/2410.23208. arXiv:2410.23208 [cs].

William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *ICML'24*, pp. 35492–35506, Vienna, Austria, July 2024. JMLR.org.

Jonas Mikhaeil, Zahra Monfared, and Daniel Durstewitz. On the difficulty of learning chaotic dynamics with RNNs. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 11297–11312. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/495e55f361708bedbab5d81f92048dcd-Paper-Conference.pdf.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and others. Human-Level Control Through Deep Reinforcement Learning. *nature*, 518(7540):529–533, 2015. Publisher: Nature Publishing Group.

Steven Morad, Ryan Kortvelesy, Matteo Bettini, Stephan Liwicki, and Amanda Prorok. POPGym: Benchmarking Partially Observable Reinforcement Learning. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=chDrutUTs0K.

Steven Morad, Ryan Kortvelesy, Stephan Liwicki, and Amanda Prorok. Reinforcement Learning with Fast and Forgetful Memory. *Advances in Neural Information Processing Systems*, 36:72008–72029, December 2023b. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/e3bf2f0f10774c474de22a12cb060e2c-Abstract-Conference.html.

Steven Morad, Chris Lu, Ryan Kortvelesy, Stephan Liwicki, Jakob Nicolaus Foerster, and Amanda Prorok. Recurrent Reinforcement Learning with Memoroids. November 2024. URL https://openreview.net/forum?id=nA4Q983a1v.

Tianwei Ni, Benjamin Eysenbach, and Ruslan Salakhutdinov. Recurrent Model-Free RL Can Be a Strong Baseline for Many POMDPs. In *Proceedings of the 39th International Conference on Machine Learning*, pp. 16691–16723. PMLR, June 2022. URL https://proceedings.mlr.press/v162/ni22a.html. ISSN: 2640-3498.

Tianwei Ni, Michel Ma, Benjamin Eysenbach, and Pierre-Luc Bacon. When Do Transformers Shine in Rl? Decoupling Memory from Credit Assignment. *Advances in Neural Information Processing Systems*, 36, 2024.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting Recurrent Neural Networks for Long Sequences. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023. Place: Honolulu, Hawaii, USA.

Christos H Papadimitriou and John N Tsitsiklis. The complexity of Markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987. Publisher: INFORMS.

Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphaël Lopez Kaufman, Aidan Clark, Seb Noury, Matthew Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing Transformers for Reinforcement Learning. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7487–7498. PMLR, July 2020. URL https://proceedings.mlr.press/v119/parisotto20a.html.

Eduardo Pignatelli, Jarek Liesen, Robert Tjarko Lange, Chris Lu, Pablo Samuel Castro, and Laura Toni. NAVIX: Scaling MiniGrid Environments with JAX, July 2024. URL http://arxiv.org/abs/2407.19396. arXiv:2407.19396 [cs].

Marco Pleines, Matthias Pallasch, Frank Zimmer, and Mike Preuss. Memory Gym: Partially Observable Challenges to Memory-Based Agents. September 2022. URL https://openreview.net/forum?id=jHc8dCx6DDr.

Marco Pleines, Matthias Pallasch, Frank Zimmer, and Mike Preuss. Memory Gym: Towards Endless Tasks to Benchmark Memory Capabilities of Agents. *Journal of Machine Learning Research*, 26(6):1–40, 2025.

Raghu Rajan, Jessica Lizeth Borja Diaz, Suresh Guttikonda, Fabio Ferreira, André Biedenkapp, Jan Ole von Hartz, and Frank Hutter. MDP Playground: An Analysis and Debug Testbed for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 77:821–890, July 2023. ISSN 1076-9757. doi: 10.1613/jair.1.14314. URL http://arxiv.org/abs/1909.07750. arXiv:1909.07750 [cs].

Mohammad Reza Samsami, Artem Zholus, Janarthanan Rajendran, and Sarath Chandar. Mastering Memory Tasks with World Models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=1vDArHJ68h.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://www.nature.com/articles/nature16961. Publisher: Nature Publishing Group.

Austin Stone, Oscar Ramirez, Kurt Konolige, and Rico Jonschkowski. The Distracting Control Suite–A Challenging Benchmark for Reinforcement Learning from Pixels. *arXiv preprint arXiv:2101.02722*, 2021.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

Ruo Yu Tao, Kaicheng Guo, Cameron Allen, and George Konidaris. Benchmarking Partial Observability in Reinforcement Learning with a Suite of Memory-Improvable Domains, July 2025. URL http://arxiv.org/abs/2508.00046. arXiv:2508.00046 [cs].

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, October 2012. doi: 10.1109/IROS.2012.6386109. URL https://ieeexplore.ieee.org/abstract/document/6386109?casa_token=1tMZ2sK7IroAAAAA:VhBdmpcQ2-7j2n2asoRU2qOZtxjiI3gcLmbFZscPxjgIhe267hBtCSJr_y__QuoXuGvfu0W-. ISSN: 2153-0866.

Edan Toledo. Stoix: Distributed Single-Agent Reinforcement Learning End-to-End in JAX, April 2024. URL https://github.com/EdanToledo/Stoix.

Nikos Vlassis, Michael L Littman, and David Barber. Stochastic POMDP controllers: How easy to optimize? In *10th European Workshop on Reinforcement Learning*, 2012.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine. June 2022. URL https://openreview.net/forum?id=BubxnHpuMbG.

Kenny Young and Tian Tian. MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments, June 2019. URL http://arxiv.org/abs/1903.03176. arXiv:1903.03176 [cs].

APPENDIX

A Environment Throughput

We examine the efficiency of our proposed environments. POPGym Arcade uses hardware acceleration for improved throughput, but large pixel observations take significant resources to render. We compare the throughput of our environments to other well-known environments in Fig. 6, using an RTX4090 GPU and Intel I7 13700 CPU. We parallelize CPU environments using synchronous VectorEnvs, and shade the 95% bootstrapped confidence interval.

Our environments are faster than all CPU-based environments we tested (Fig. 6). Our observations are about 10,000 times larger than POPGym, yet we run approximately 100 times faster. We achieve similar throughput to hardware-accelerated Gymnax while generating observations four orders of magnitude larger. One can quickly prototype models or algorithms on our pixel-space tasks.

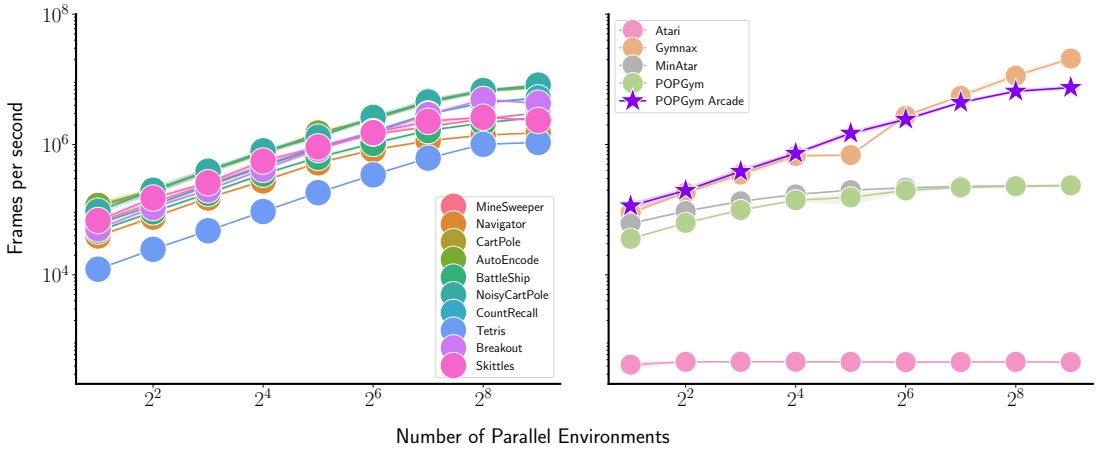


Figure 6: We plot environment throughput and compare to prior work. We achieve linear scaling until 2^7 environments, after which we saturate GPU cores. POPGym Arcade is faster than Atari, and outperforms POPGym and MinAtar while producing observations orders of magnitude larger.

B Recall Density Analysis by Model and Task

In this section, we plot the recall density after training, for each model and task. We discretize τ into sixteen bins and compute the density over five random seeds. No matter the memory model, uninformative prior observations still tend to affect future decisions in MDPs. Even in the best cases, much of the density in MDPs is distributed across old observations (Fig. 9).

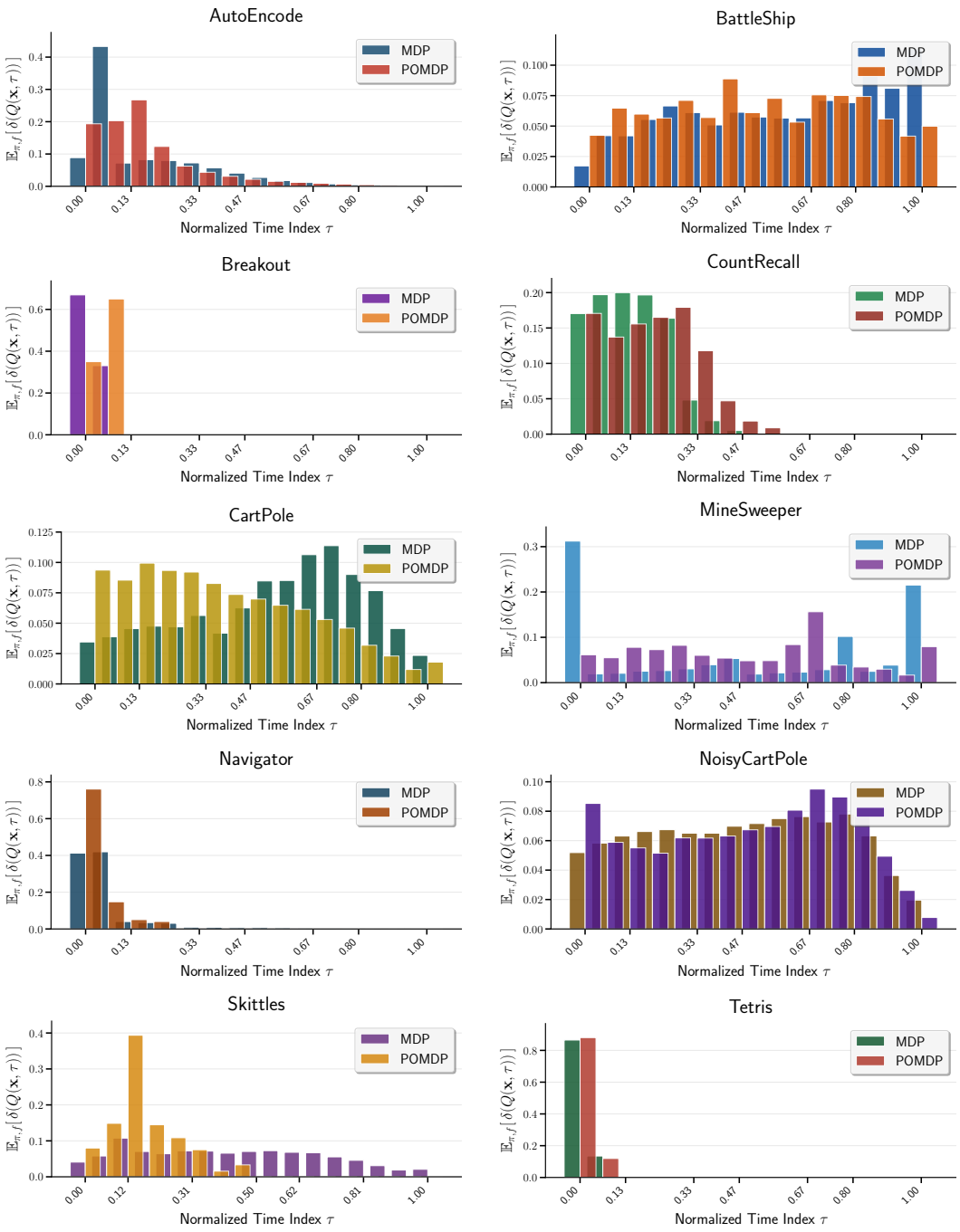


Figure 7: Recall density of FART categorized by environment, over five random seeds.

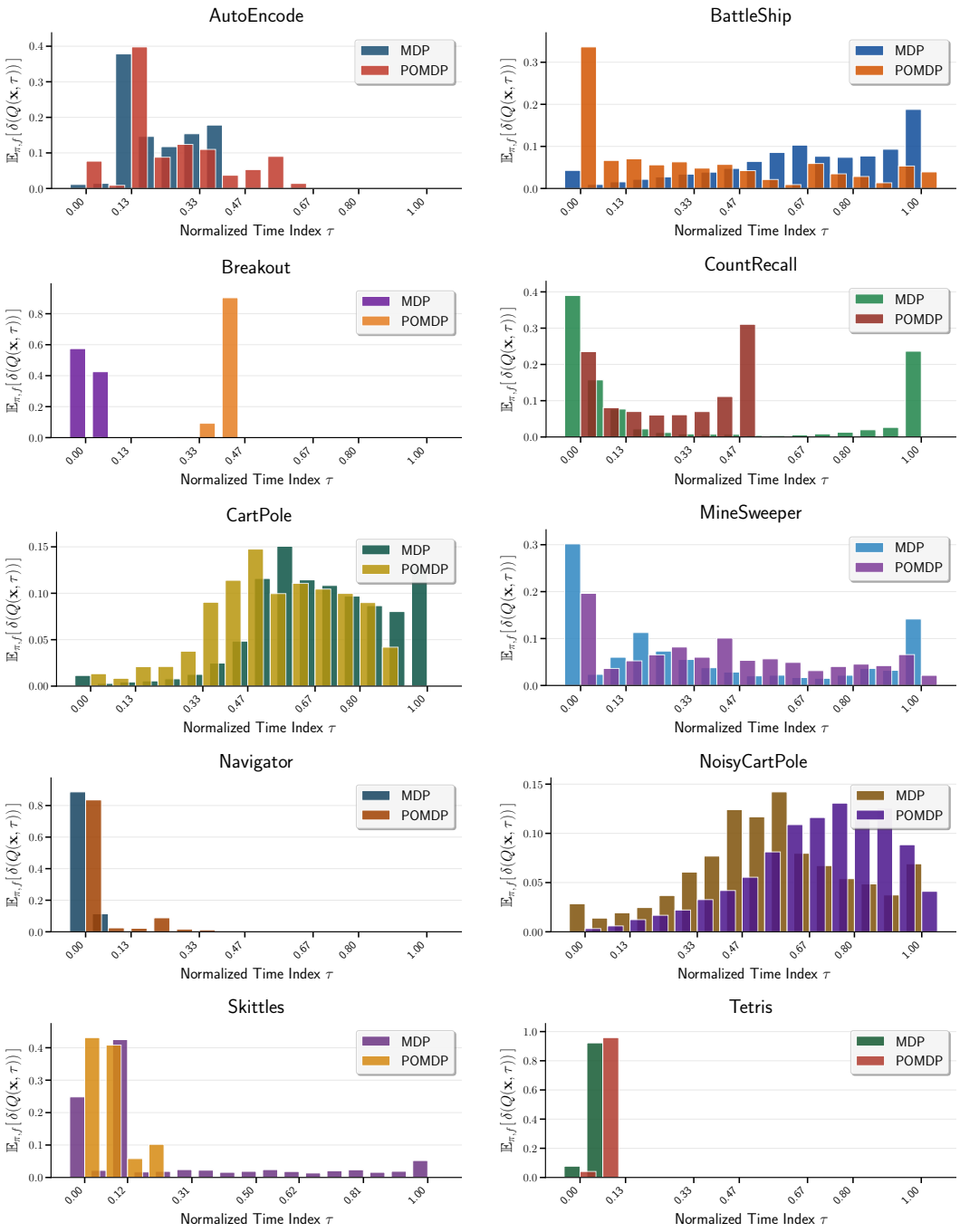


Figure 8: Recall density of GRU categorized by environment, over five random seeds.

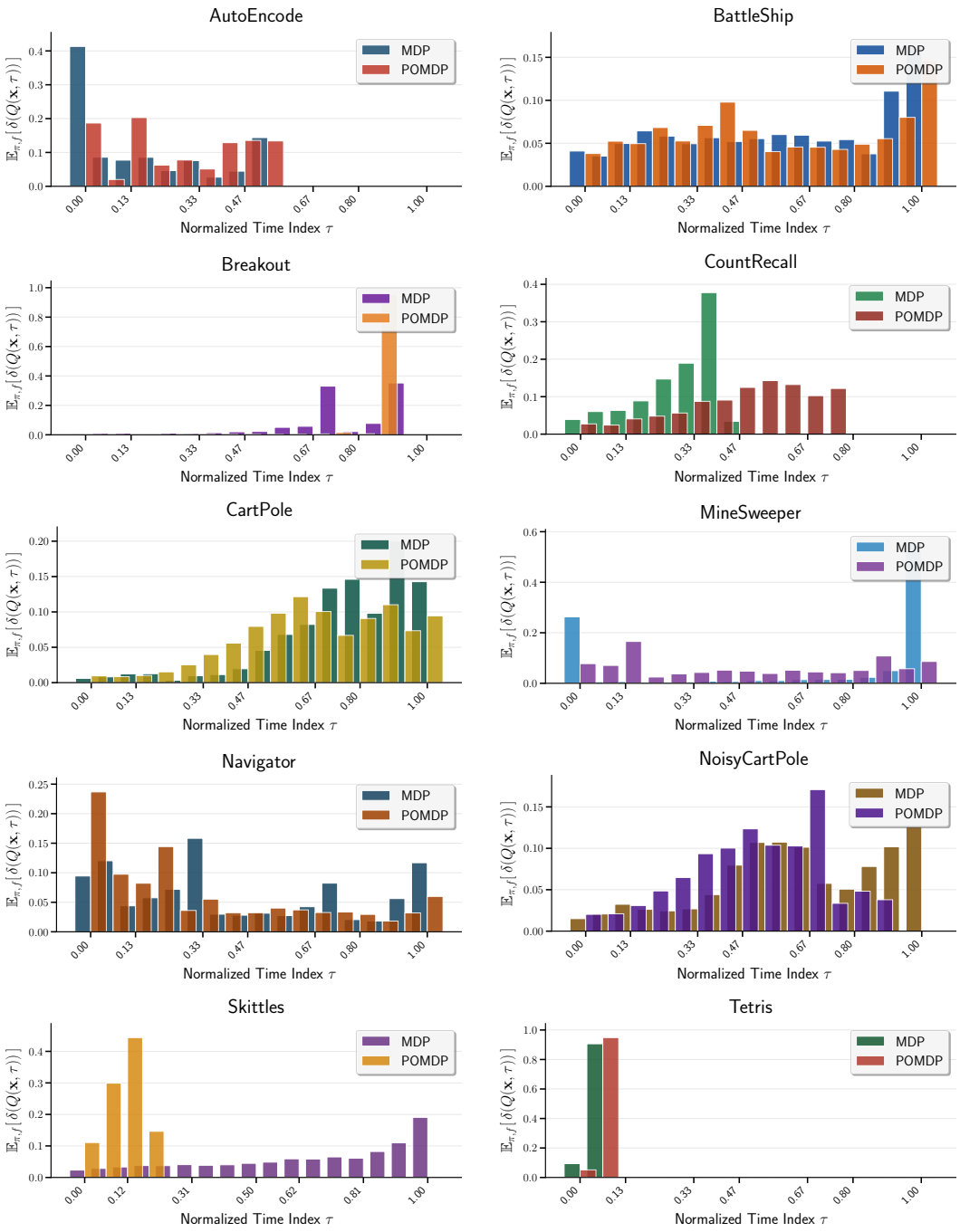


Figure 9: Recall density of LRU categorized by environment, over five random seeds.

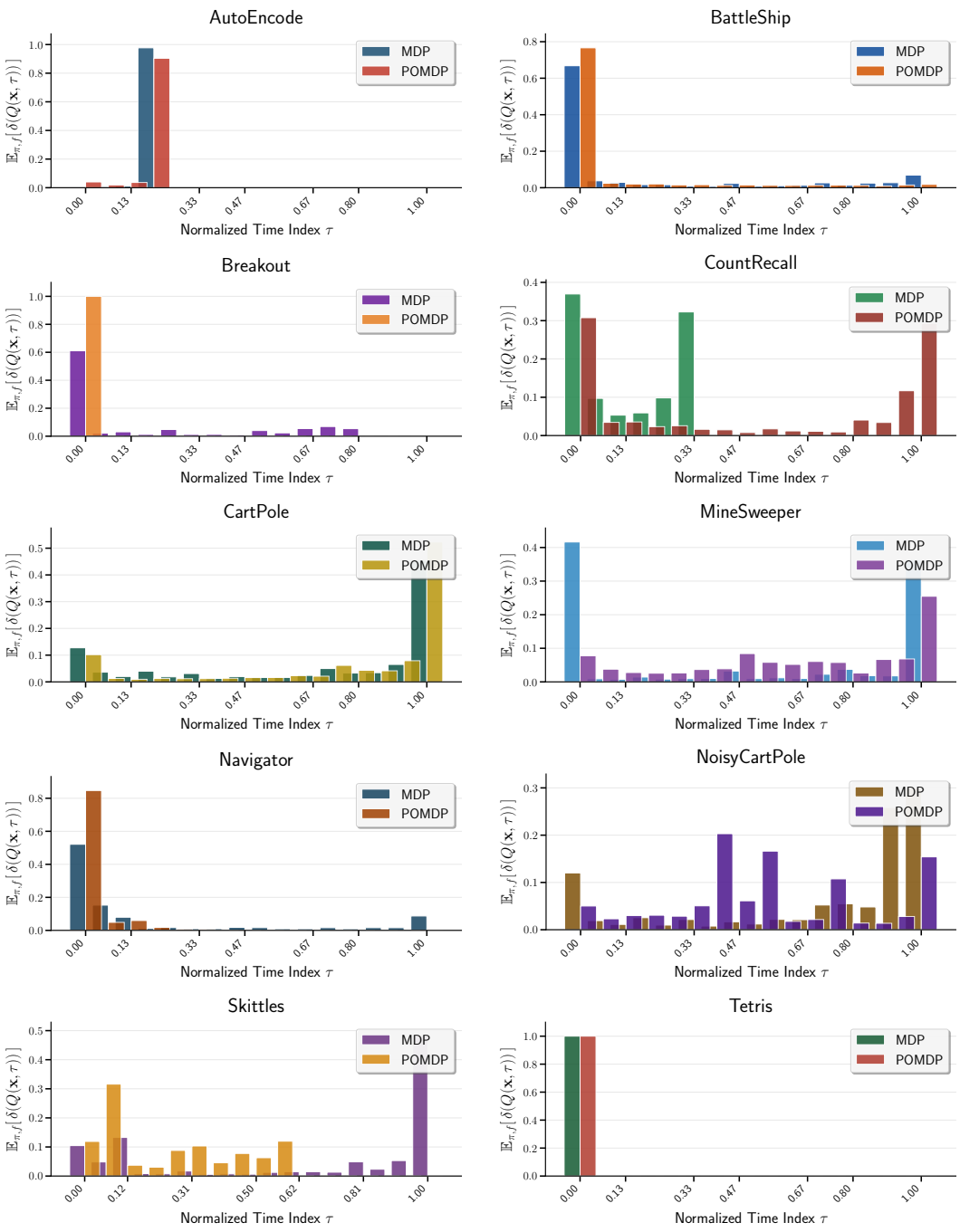


Figure 10: Recall density of MinGRU categorized by environment, over five random seeds.

C Additional Pixel-Level Memory Analysis Experiments

We provide further pixel-level analysis of our environment is using Eq. (4). We see that memory models ignore Markov observations, still storing and recalling information in the MDP case.

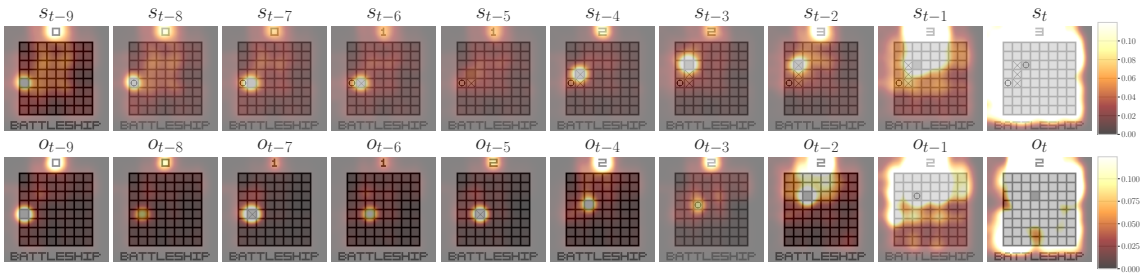


Figure 11: LRU saliency on the BattleShip task, plotted via Eq. (4) using the L2 norm. The top row represents the MDP and the bottom row represents the equivalent POMDP.

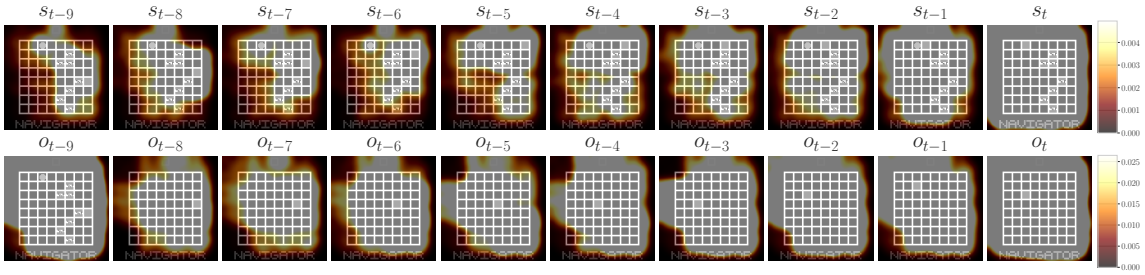


Figure 12: LRU saliency on the Navigator task, plotted via Eq. (4) using the L1 norm. The top row represents the MDP and the bottom row represents the equivalent POMDP.

D PPO AND DQN EXPERIMENTS

PQN is as close as we can possibly get to pure deep value function approximation, discarding replay buffers, target networks, and so on. However, it is not clear if our PQN findings extend to policy gradient algorithms (PPO) or algorithms with experience replay (DQN). In this section, we repeat our PQN analyses for the DQN and PPO algorithms, demonstrating our results generally hold across other RL algorithms as well. Specifically, we aim to demonstrate the generalizability and applicability of Conclusion (**C3**), (**C4**) and (**C5**).

D.1 MODEL-BASED REWARD PREDICTORS

To simulate a mechanism analogous to model-based prediction, we set the discount factor γ to 0 in the PQN experiment. This modification restricts the Q-function to learning only the immediate reward, thereby decoupling it from long-term cumulative returns. From the figure Fig. 13, we can clearly observe that value smearing is still present.

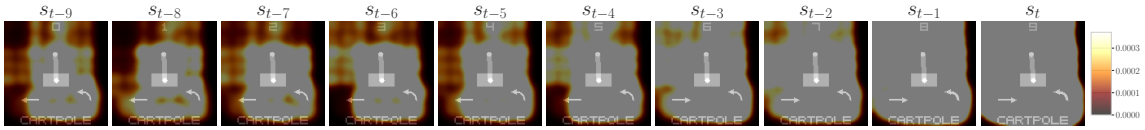


Figure 13: LRU saliency on the CartPole MDP task, with PQN and $\gamma = 0$. Setting the discount factor γ to 0 is intended to simulate a model-based mechanism, allowing for analysis of how the agent uses historical information to evaluate immediate states or rewards.

D.2 PPO

To further verify the generalizability of our findings, we extended our experiments to PPO and analyzed the policy's memory utilization. In Fig. 14, We introduced our new metrics (Definition 4.1, Definition 4.2) for PPO. Figure 15 and Fig. 16 indicate that our findings extend to policy optimization, where the issue persists but is replaced by policy smearing. Similarly, Fig. 17 provides evidence that the phenomenon described in Item (**C5**) also occurs in PPO.

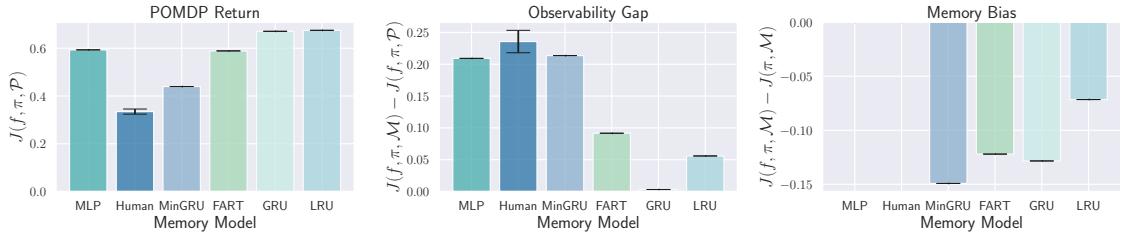


Figure 14: Disentangling returns for the PPO algorithm. We replicate the analysis from Fig. 2 with PPO to verify the generalizability of our findings. We plot the aggregated POMDP returns, Observability Gap (Definition 4.1), and Memory Bias (Definition 4.2) across all environments and the Easy difficulty configuration. The results confirm that the negative memory bias and performance gaps observed in value-based methods (PQN) persist in policy optimization settings.

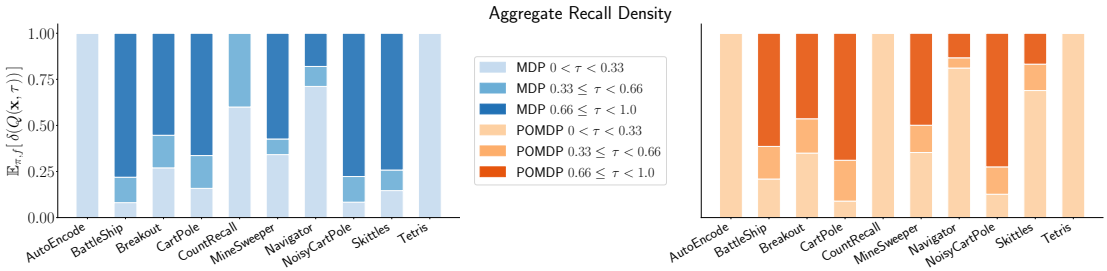


Figure 15: Aggregate Recall Density for PPO on Easy task variants. We plot the expected contribution of historical observations to the value estimate. The results confirm that policy smearing exists in PPO.

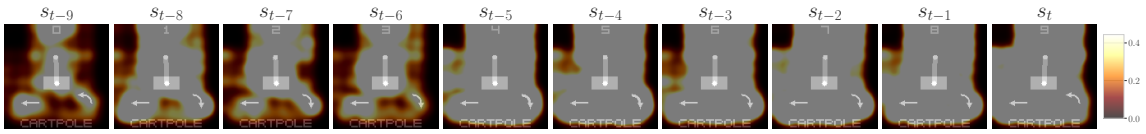


Figure 16: Pixel-level saliency for PPO with LRU on CartPole MDP. We visualize gradients of the policy logits with respect to past observations. The results demonstrate that policy smearing occurs in PPO method.

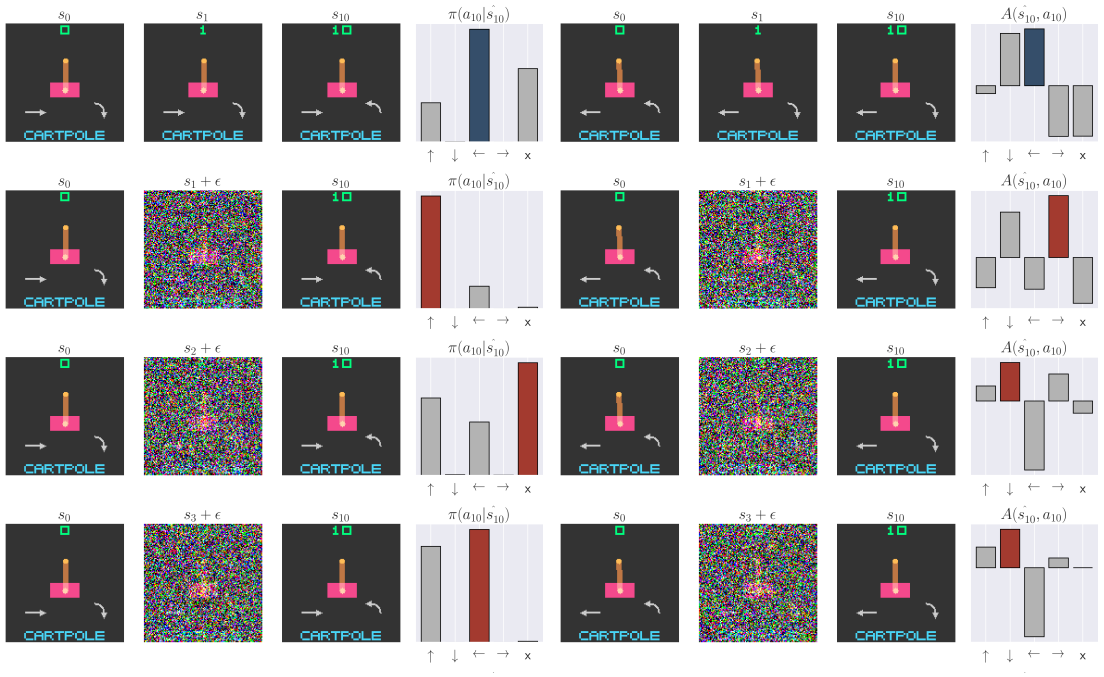


Figure 17: Noise injection experiments on additional baselines. **(Left) PPO with LRU**, we plot the policy distribution $\pi(a|s)$ after injecting noise, observing results similar to PQN. **(Right) DQN with FART**, we illustrate the relative Q values (A) after noise injection. Both experiments confirm that the vulnerability to state contamination persists across different methods and models.

D.3 DQN

We extended our verification to RL algorithms equipped with replay buffers. Specifically, we evaluated DQN in three representative environments, with the supporting evidence presented in Fig. 18, Fig. 19, Fig. 20 and Fig. 17.

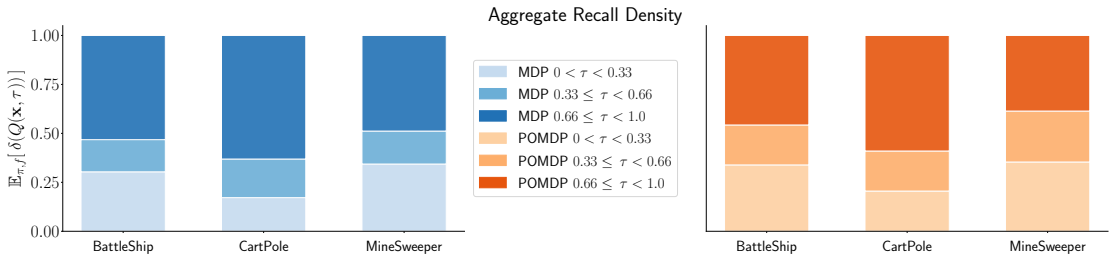


Figure 18: Aggregate Recall Density for DQN. We evaluate DQN on three representative Easy tasks. The results confirm that value smearing persists even with replay buffers.

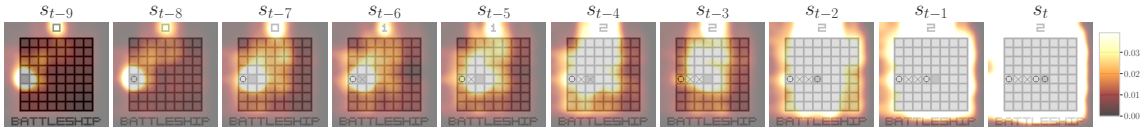


Figure 19: Pixel-level saliency for DQN with LRU on BattleShip MDP. We visualize the gradient of the Q-value with respect to past observations. Consistent with our PQN results, we observe value smearing even in the off-policy DQN setting.

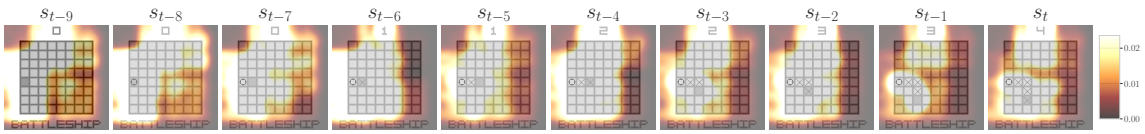


Figure 20: Pixel-level saliency for DQN with FART on BattleShip MDP.

E Additional Recurrent State Contamination Experiments

We provide additional experiments for recurrent policy contamination. We use the cartpole example for even longer rollout horizons of 50 and 100 timesteps. We find that old observations still have a significant impact on the relative Q values (A) and corresponding actions.

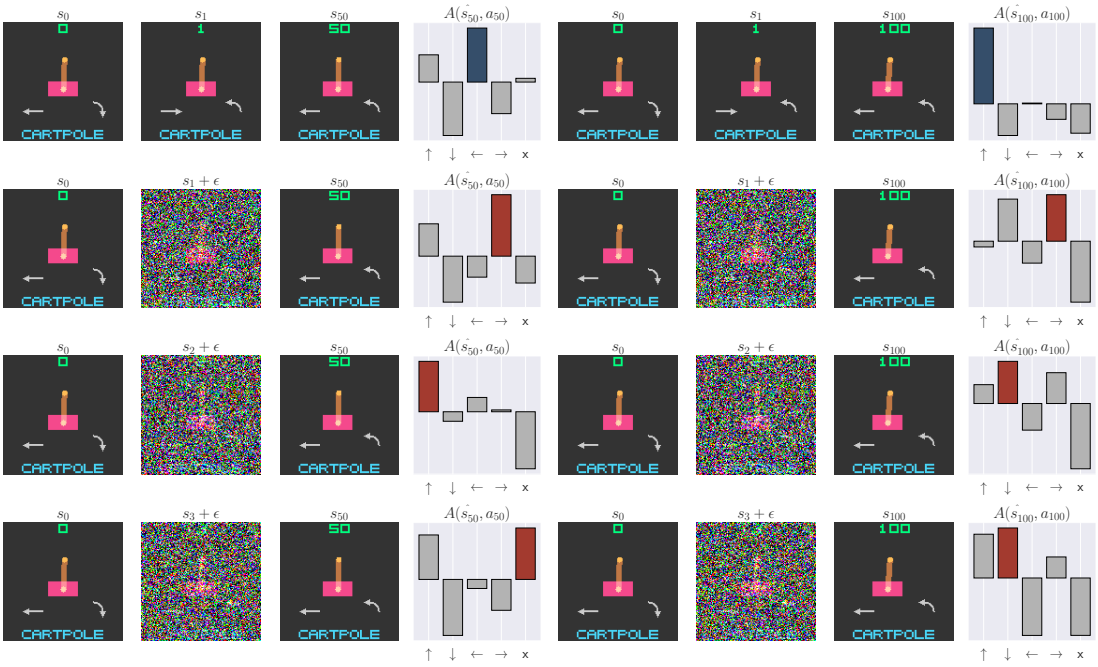


Figure 21: We demonstrate recurrent state contamination over longer horizons, using the LRU model. At 50 and 100 timesteps in the future, corrupted observations can still influence relative Q values (A) enough to change the selected action.

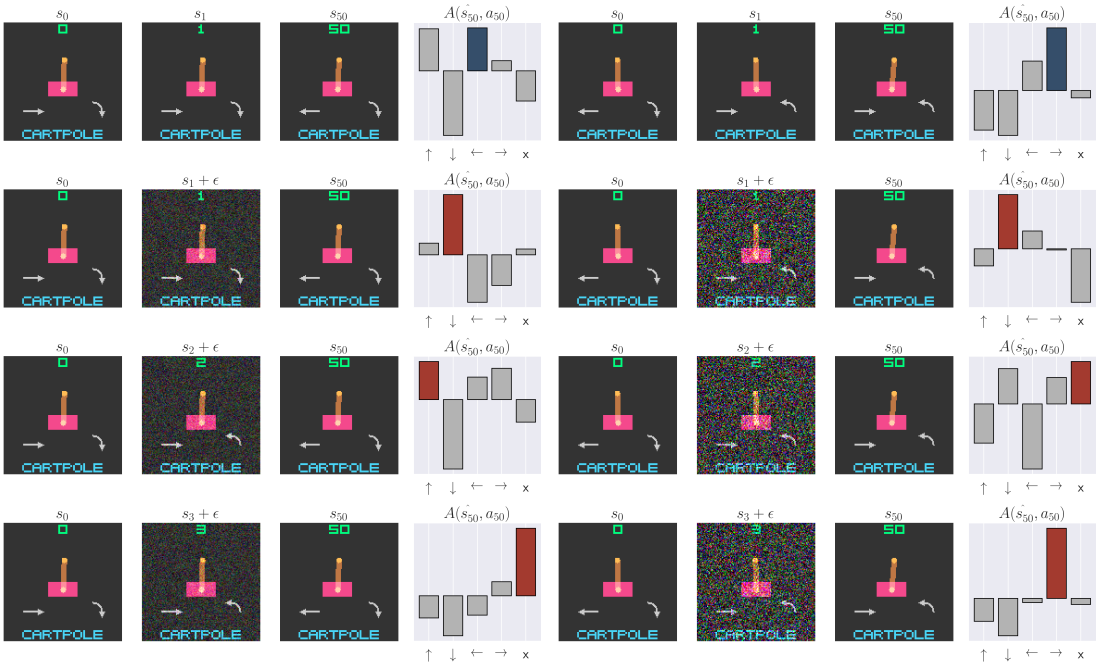
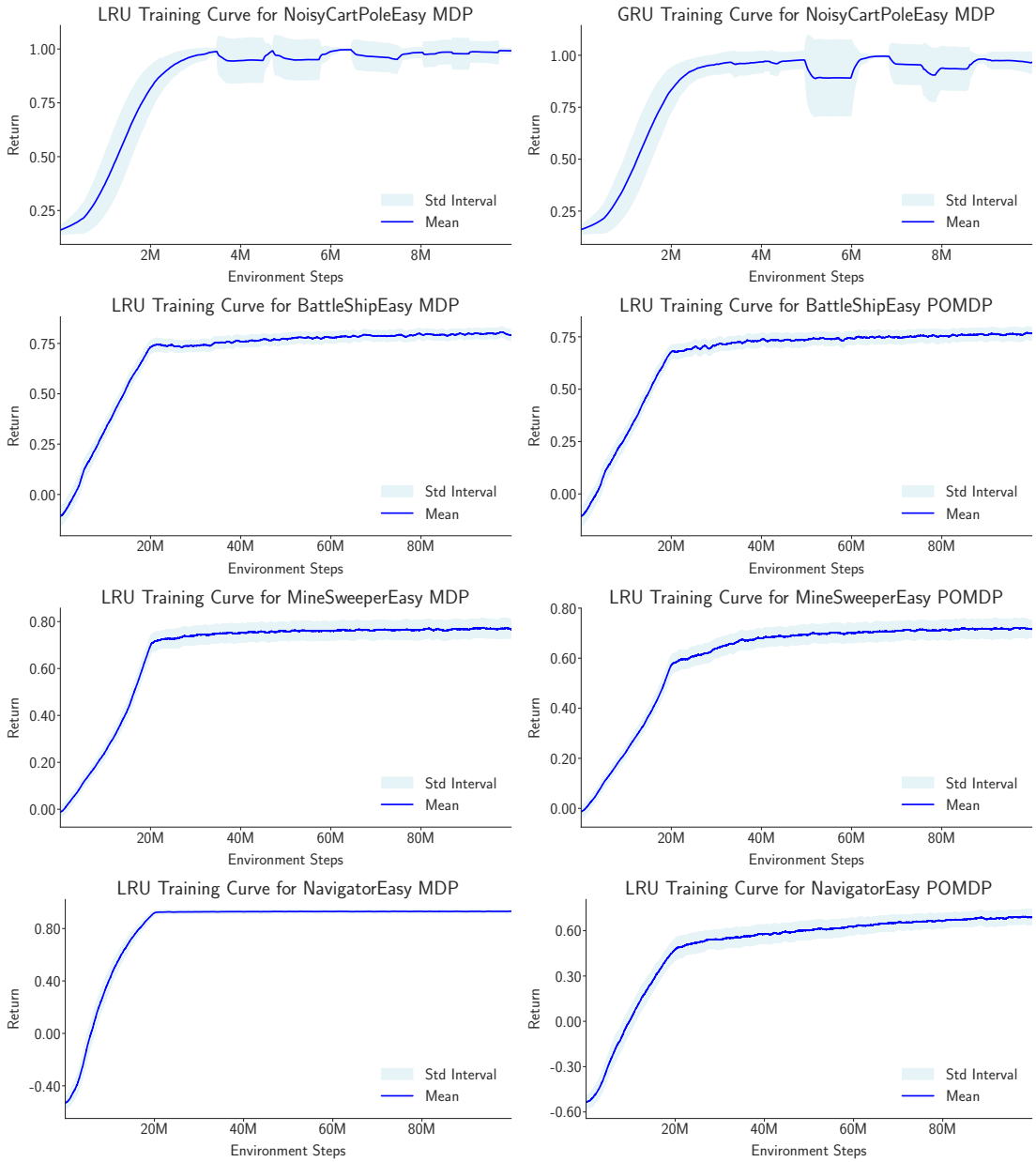


Figure 22: We prove that when using different levels of noise injection, the obtained relative Q values (A) will still be affected.

F Pixel Gradient Training Curves

We plot the return curves for our pixel gradient analysis experiments. We train until the MDPs appear converged.



G Return Analysis by Model

In this section, we provide the unnormalized returns (mean and standard deviation) for the aggregated results in Fig. 2. We provide values for each environment, memory model, and difficulty level.

Table 2: Return for the LRU model, broken down by task and observability.

Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.26 ± 0.00	0.27 ± 0.01
AutoEncodeMedium	0.24 ± 0.01	0.24 ± 0.00
AutoEncodeHard	0.23 ± 0.02	0.22 ± 0.02
BattleShipEasy	0.85 ± 0.02	0.84 ± 0.01
BattleShipMedium	0.78 ± 0.03	0.82 ± 0.01
BattleShipHard	0.73 ± 0.04	0.80 ± 0.01
BreakoutEasy	0.89 ± 0.01	0.75 ± 0.04
BreakoutMedium	0.83 ± 0.02	0.76 ± 0.06
BreakoutHard	0.84 ± 0.00	0.72 ± 0.09
CartPoleEasy	0.99 ± 0.00	0.99 ± 0.00
CartPoleMedium	0.98 ± 0.01	0.98 ± 0.01
CartPoleHard	0.97 ± 0.01	0.95 ± 0.02
CountRecallEasy	0.47 ± 0.06	0.36 ± 0.01
CountRecallMedium	0.22 ± 0.03	0.23 ± 0.04
CountRecallHard	0.15 ± 0.02	0.23 ± 0.01
MineSweeperEasy	0.87 ± 0.00	0.82 ± 0.01
MineSweeperMedium	0.67 ± 0.01	0.66 ± 0.01
MineSweeperHard	0.62 ± 0.01	0.61 ± 0.00
NavigatorEasy	0.91 ± 0.01	0.44 ± 0.09
NavigatorMedium	0.91 ± 0.01	0.32 ± 0.03
NavigatorHard	0.94 ± 0.01	0.40 ± 0.06
NoisyCartPoleEasy	0.99 ± 0.00	0.99 ± 0.00
NoisyCartPoleMedium	0.98 ± 0.01	0.99 ± 0.00
NoisyCartPoleHard	0.97 ± 0.01	0.97 ± 0.01
SkittlesEasy	0.83 ± 0.01	0.52 ± 0.02
SkittlesMedium	0.79 ± 0.01	0.51 ± 0.00
SkittlesHard	0.74 ± 0.01	0.47 ± 0.00
TetrisEasy	0.01 ± 0.00	0.01 ± 0.00
TetrisMedium	0.01 ± 0.00	0.01 ± 0.00
TetrisHard	0.01 ± 0.00	0.01 ± 0.00

Table 3: Return for the MinGRU model, broken down by task and observability.

Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.27 ± 0.00	0.27 ± 0.00
AutoEncodeMedium	0.25 ± 0.01	0.26 ± 0.00
AutoEncodeHard	0.25 ± 0.00	0.25 ± 0.00
BattleShipEasy	0.82 ± 0.02	0.76 ± 0.01
BattleShipMedium	0.73 ± 0.02	0.75 ± 0.02
BattleShipHard	0.71 ± 0.03	0.73 ± 0.02
BreakoutEasy	0.88 ± 0.02	0.36 ± 0.02
BreakoutMedium	0.83 ± 0.01	0.33 ± 0.04
BreakoutHard	0.76 ± 0.03	0.28 ± 0.02
CartPoleEasy	0.98 ± 0.01	0.87 ± 0.02
CartPoleMedium	0.93 ± 0.02	0.68 ± 0.06
CartPoleHard	0.88 ± 0.08	0.54 ± 0.03
CountRecallEasy	0.31 ± 0.01	0.29 ± 0.03
CountRecallMedium	0.17 ± 0.00	0.21 ± 0.04
CountRecallHard	0.15 ± 0.01	0.13 ± 0.04
MineSweeperEasy	0.83 ± 0.01	0.76 ± 0.01
MineSweeperMedium	0.64 ± 0.00	0.61 ± 0.00
MineSweeperHard	0.58 ± 0.01	0.58 ± 0.01
NavigatorEasy	0.89 ± 0.02	0.33 ± 0.08
NavigatorMedium	0.74 ± 0.08	0.25 ± 0.12
NavigatorHard	0.13 ± 0.45	0.09 ± 0.08
NoisyCartPoleEasy	0.95 ± 0.01	0.89 ± 0.01
NoisyCartPoleMedium	0.92 ± 0.01	0.86 ± 0.01
NoisyCartPoleHard	0.86 ± 0.02	0.81 ± 0.01
SkittlesEasy	0.80 ± 0.00	0.39 ± 0.01
SkittlesMedium	0.76 ± 0.02	0.36 ± 0.01
SkittlesHard	0.73 ± 0.01	0.31 ± 0.01
TetrisEasy	0.01 ± 0.00	0.01 ± 0.00
TetrisMedium	0.01 ± 0.01	0.01 ± 0.01
TetrisHard	0.01 ± 0.00	0.00 ± 0.00

Table 4: Return for the GRU model, broken down by task and observability.

Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.26 ± 0.00	0.26 ± 0.00
AutoEncodeMedium	0.23 ± 0.01	0.24 ± 0.02
AutoEncodeHard	0.23 ± 0.01	0.23 ± 0.02
BattleShipEasy	0.84 ± 0.01	0.84 ± 0.01
BattleShipMedium	0.76 ± 0.05	0.81 ± 0.01
BattleShipHard	0.71 ± 0.04	0.82 ± 0.01
BreakoutEasy	0.89 ± 0.01	0.62 ± 0.10
BreakoutMedium	0.85 ± 0.01	0.60 ± 0.08
BreakoutHard	0.85 ± 0.02	0.44 ± 0.02
CartPoleEasy	1.00 ± 0.00	0.98 ± 0.01
CartPoleMedium	0.99 ± 0.01	0.94 ± 0.02
CartPoleHard	0.96 ± 0.01	0.93 ± 0.04
CountRecallEasy	0.37 ± 0.04	0.36 ± 0.01
CountRecallMedium	0.19 ± 0.02	0.21 ± 0.03
CountRecallHard	0.16 ± 0.04	0.18 ± 0.08
MineSweeperEasy	0.85 ± 0.01	0.81 ± 0.01
MineSweeperMedium	0.66 ± 0.00	0.65 ± 0.01
MineSweeperHard	0.61 ± 0.01	0.61 ± 0.00
NavigatorEasy	0.90 ± 0.01	0.44 ± 0.06
NavigatorMedium	0.75 ± 0.25	0.34 ± 0.09
NavigatorHard	0.90 ± 0.04	0.38 ± 0.06
NoisyCartPoleEasy	0.98 ± 0.01	0.99 ± 0.00
NoisyCartPoleMedium	0.97 ± 0.02	0.98 ± 0.01
NoisyCartPoleHard	0.95 ± 0.02	0.97 ± 0.02
SkittlesEasy	0.86 ± 0.01	0.50 ± 0.00
SkittlesMedium	0.83 ± 0.00	0.45 ± 0.00
SkittlesHard	0.75 ± 0.00	0.40 ± 0.00
TetrisEasy	0.01 ± 0.00	0.01 ± 0.00
TetrisMedium	0.01 ± 0.00	0.01 ± 0.00
TetrisHard	0.01 ± 0.00	0.01 ± 0.00

Table 5: Return for the FART model, broken down by task and observability.

Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.25 ± 0.00	0.26 ± 0.00
AutoEncodeMedium	0.23 ± 0.01	0.23 ± 0.01
AutoEncodeHard	0.20 ± 0.03	0.21 ± 0.02
BattleShipEasy	0.66 ± 0.09	0.77 ± 0.02
BattleShipMedium	0.51 ± 0.01	0.52 ± 0.02
BattleShipHard	0.50 ± 0.00	0.50 ± 0.00
BreakoutEasy	0.10 ± 0.04	0.13 ± 0.02
BreakoutMedium	0.14 ± 0.01	0.11 ± 0.05
BreakoutHard	0.13 ± 0.01	0.10 ± 0.05
CartPoleEasy	0.98 ± 0.01	0.97 ± 0.01
CartPoleMedium	0.85 ± 0.06	0.78 ± 0.05
CartPoleHard	0.62 ± 0.06	0.60 ± 0.04
CountRecallEasy	0.39 ± 0.03	0.37 ± 0.02
CountRecallMedium	0.06 ± 0.00	0.10 ± 0.04
CountRecallHard	0.06 ± 0.01	0.05 ± 0.00
MineSweeperEasy	0.76 ± 0.01	0.73 ± 0.04
MineSweeperMedium	0.61 ± 0.02	0.59 ± 0.02
MineSweeperHard	0.55 ± 0.00	0.55 ± 0.01
NavigatorEasy	0.46 ± 0.14	0.36 ± 0.05
NavigatorMedium	-0.27 ± 0.19	-0.28 ± 0.19
NavigatorHard	-0.45 ± 0.03	-0.41 ± 0.02
NoisyCartPoleEasy	0.98 ± 0.01	0.97 ± 0.02
NoisyCartPoleMedium	0.96 ± 0.02	0.96 ± 0.01
NoisyCartPoleHard	0.92 ± 0.02	0.89 ± 0.01
SkittlesEasy	0.85 ± 0.01	0.32 ± 0.02
SkittlesMedium	0.80 ± 0.01	0.29 ± 0.01
SkittlesHard	0.75 ± 0.00	0.28 ± 0.01
TetrisEasy	0.01 ± 0.00	0.01 ± 0.00
TetrisMedium	0.01 ± 0.01	0.01 ± 0.01
TetrisHard	0.01 ± 0.00	0.00 ± 0.00

Table 6: Return for the MLP model, broken down by task and observability.

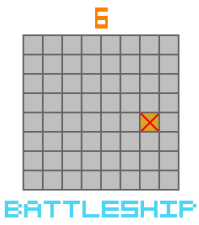
Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.26 ± 0.00	0.27 ± 0.00
AutoEncodeMedium	0.25 ± 0.00	0.25 ± 0.00
AutoEncodeHard	0.25 ± 0.00	0.25 ± 0.00
BattleShipEasy	0.85 ± 0.02	0.60 ± 0.04
BattleShipMedium	0.70 ± 0.04	0.56 ± 0.01
BattleShipHard	0.75 ± 0.03	0.53 ± 0.00
BreakoutEasy	0.93 ± 0.01	0.49 ± 0.02
BreakoutMedium	0.89 ± 0.02	0.49 ± 0.06
BreakoutHard	0.84 ± 0.02	0.34 ± 0.05
CartPoleEasy	0.99 ± 0.00	0.86 ± 0.02
CartPoleMedium	0.96 ± 0.01	0.47 ± 0.01
CartPoleHard	0.95 ± 0.01	0.31 ± 0.01
CountRecallEasy	0.40 ± 0.04	0.15 ± 0.01
CountRecallMedium	0.17 ± 0.02	0.06 ± 0.00
CountRecallHard	0.15 ± 0.01	0.08 ± 0.00
MineSweeperEasy	0.86 ± 0.01	0.72 ± 0.01
MineSweeperMedium	0.65 ± 0.01	0.59 ± 0.00
MineSweeperHard	0.57 ± 0.00	0.55 ± 0.00
NavigatorEasy	0.88 ± 0.02	-0.20 ± 0.05
NavigatorMedium	0.86 ± 0.01	-0.19 ± 0.02
NavigatorHard	0.94 ± 0.01	-0.07 ± 0.08
NoisyCartPoleEasy	0.94 ± 0.01	0.82 ± 0.02
NoisyCartPoleMedium	0.86 ± 0.01	0.74 ± 0.01
NoisyCartPoleHard	0.70 ± 0.01	0.66 ± 0.01
SkittlesEasy	0.84 ± 0.00	0.41 ± 0.00
SkittlesMedium	0.81 ± 0.00	0.37 ± 0.00
SkittlesHard	0.75 ± 0.01	0.33 ± 0.00
TetrisEasy	0.39 ± 0.02	0.39 ± 0.02
TetrisMedium	0.39 ± 0.02	0.39 ± 0.02
TetrisHard	0.22 ± 0.01	0.22 ± 0.01

H Environment Descriptions

We describe each of our implemented MDPs and rules. Then, we explain how and why we make each MDP into a POMDP.

H.1 Battleship

MDP The agent's goal is to sink all the pre-placed ships to win the game. The agent controls a cursor that represents its current position on the grid. Each turn, the agent can either move the cursor one space or choose to "FIRE" the current grid cell. Each cell can be in one of three states: COVERED, HIT, or MISS. Initially, all cells are COVERED, but each changes to HIT or MISS upon firing. The agent receives a positive reward for hitting a ship, a neutral reward (0) for the first miss on a COVERED cell, and a negative reward for repeatedly firing HIT or MISS cells.

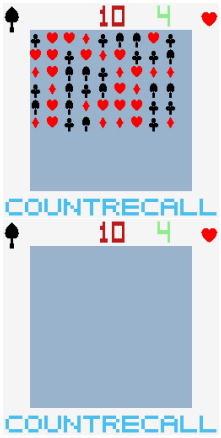


POMDP We make this task partially observable by only showing the HIT or MISS markers for the tile the cursor currently occupies. All other tiles appear COVERED, regardless of whether they have been fired upon. This does not affect the reward function, only the observation function.

POMDP Required Capabilities The agent must remember the tile label (HIT/MISS) of previously fired upon tiles, or risk running out of moves. This capability is critical for avoiding redundant actions, which incur negative rewards, and performing an efficient search of locations and ships. This will test spatial memory and integrate a sequence of localized observations into a coherent ability.

MDP Recovery from POMDP By retaining all prior observations, the agent can view all the same HIT/MISS markers as in the MDP.

H.2 Count Recall



MDP Every turn, the agent gets a value card and a query card. All previous value cards are displayed to the agent. The agent's task is to figure out how many times the query card has shown up so far. To do this, the agent counts how many times it has received the matching value card and uses that as the answer. If the agent guesses the correct count, it receives a positive reward. If the guess is incorrect, the agent receives no reward (0). This setup encourages the agent to accurately track and recall the frequency of specific cards over time.

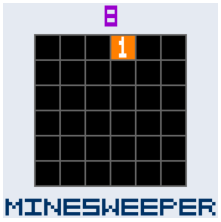
POMDP All previous value cards are hidden, the agent may only observe the current value and query card.

POMDP Required Capabilities The agent must learn to implement a latent counter for each card type to track the number of times a card has appeared. At each timestep, the agent must utilize these counters to answer the query.

MDP Recovery from POMDP By retaining all prior observations, the agent can reconstruct the MDP view from the suits at the top of the screen.

H.3 MINE SWEEPER

MDP The agent's goal is to hit all the non-mine cells to win the game, inspired by the classic computer game Minesweeper. Similar to Battleship environments, the agent controls a cursor to navigate the grid and can either move to an adjacent cell or sweep the current cell each turn. Sweeping a safe cell earns the agent a positive reward and reveals the number of adjacent mines, while hitting a mine results in negative reward and ends the game. If the agent tries to sweep a cell that has already been revealed, it receives a small negative reward. This reward structure encourages the agent to explore efficiently, avoid mines, and minimize redundant actions to maximize its cumulative reward.

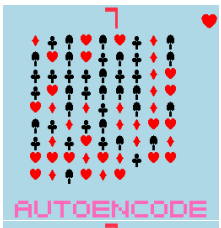


POMDP In the POMDP variant, the agent can only observe the tile at the cursor position. All other tiles appear covered.

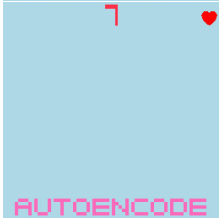
POMDP Required Capabilities The agent must remember the number of adjacent mines for previously swept tiles, while simultaneously learning the fairly complex rules of the game. In particular, the agent must predict the location of mines from a partial view of adjacency information, which itself must be memorized.

MDP Recovery from POMDP By retaining all prior observations, the agent can view all the same tile values as in the MDP.

H.4 AUTOENCODE



MDP This game is similar to Simon but played in reverse. The agent's task is to recall and replay a sequence of cards in the opposite order they were shown. During the WATCH phase, the agent observes a randomly generated sequence of cards. This sequence is displayed on-screen. In the PLAY phase, the agent must reproduce the sequence in reverse order. The agent receives a positive reward for each correct card played and no reward (0) for incorrect choices.

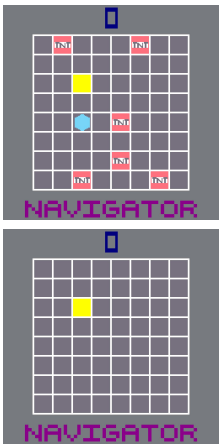


POMDP This variant does not display the series of cards to the screen. At each timestep during the WATCH phase, the agent receives only the corresponding card.

POMDP Required Capabilities By playing the cards in reverse order, the agent must learn to push and pop from a stack in latent space. Sequence permutation (pushing and popping) is well-known to be NC^1 complexity and not solvable by transformers in constant depth (Merrill et al., 2024). Only nonlinear RNNs can solve such a task in constant depth.

MDP Recovery from POMDP By retaining all prior observations, the agent can reconstruct the MDP view using the suits at the top of the screen.

H.5 NAVIGATOR



MDP The agent's goal is to navigate and open the treasure on the board while avoiding the TNT. The agent navigates using a cursor that marks its current position. On each turn, it can either shift the cursor to an adjacent cell or decide to open the cell it is currently in. Every move the agent makes results in a small negative reward. If the agent lands on a TNT cell, it receives a significant negative reward, and the game ends immediately. This environment challenges the agent to discover the most efficient path to the treasure, aiming to maximize its cumulative reward by completing the game as quickly as possible.

POMDP The agent only sees the position of the treasure and TNT blocks at the initial timestep. Afterwards, the agent can only see its current position.

POMDP Required Capabilities The agent must remember the position of the treasure and TNT blocks over time. It must perform path planning within the learned memory state to reach the goal quickly.

MDP Recovery from POMDP By retaining the first observation, the agent has access to the goal location. Retaining all consecutive actions enables reconstruction of the agent's current location.

H.6 SKITTLES

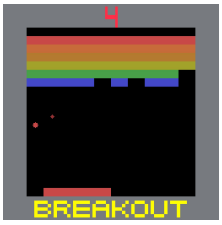
MDP This game is inspired by many Atari games that focus on moving the main character to avoid touching enemies. The agent is a white square at the bottom of the screen and must dodge colored falling blocks. Upon touching a colored block, the game terminates. The agent receives a positive reward for surviving and a negative reward for touching a colored block. We spawn the colored blocks in such a way that the agent always has a feasible path for survival.

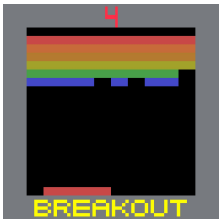
POMDP In the POMDP variant, each colored block has only a 50% probabilty of being rendered at each timestep, causing them to flicker in and our of the agent's view, the agent cannot rely on the current observation to navigate safely.

POMDP Required Capabilities The agent must use its memory to develop a form of object permanence, tracking the downward trajectories of blocks even when they are temporarily invisible. This requires the agent to maintain an internal belief about all colored blocks, integrating history observations to infer the complete observation of the envvironment.

MDP Recovery from POMDP The blocks have a 50% chance of rendering. Given that there are 11 tiles between the block spawn point and the agent, there is a $1 - 0.5^{11} = 0.9995$ chance that each block will be rendered at least once. By retaining all frames, the agent can reconstruct the position of all blocks (MDP state).

H.7 BREAKOUT





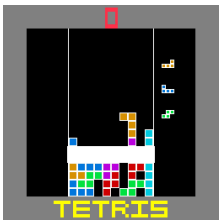
MDP This game is based on the classic Atari Breakout game. The agent controls a paddle that they may use to deflect a moving ball. When the ball touches colored blocks, the blocks deflect the ball and then disappear. The agent receives positive reward when the ball collides with colored blocks, and a negative reward and termination condition when the ball passes the paddle and leaves the bottom of the play area. To make this game fully observable, we also provide a ball "tail" that determines the velocity direction and magnitude of the ball. Upon clearing all the blocks, the game terminates.

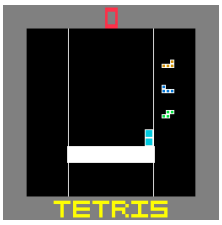
POMDP In the POMDP variant, we remove the tail. Furthermore, the ball is only visible when moving upward. It becomes invisible when moving downward. We make sure to start the game with the ball moving upward so the agent knows the initial position and velocity of the ball.

POMDP Required Capabilities Unlike other tasks, Breakout enables episodes that are thousands of timesteps long, allowing us to understand how memory adapts over very long sequences. This POMDP primarily tests short-term memory, as the agent need only (1) integrate position to predict velocity and (2) remember where the ball was a few timesteps ago to predict the future path of the ball. But it must learn a representation that can do so reliably over long durations.

MDP Recovery from POMDP By retaining all observations up to and including collision of the ball with the block, the agent can predict the deterministic angle and velocity at which the ball will reflect. This information is sufficient to predict the position and velocity of the ball, recovering the MDP.

H.8 TETRIS





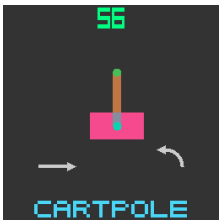
MDP This is the well-known Russian puzzle game. Colored blocks of various shapes are dropped from the top of the screen, and the player controls their descent (position and rotation). Upon reaching the bottom of the screen or touching another block, the controlled block is frozen in-place and a new agent-controlled block is spawned. The blocks slowly pile up, and the agent receives a negative reward and termination upon the blocks reaching the top of the screen. Completely filling a row with blocks "clears" the row, deleting the tiles in the row and providing a positive reward. The game also terminates after clearing a predetermined number of rows.

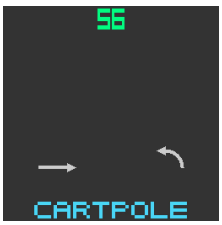
POMDP In the POMDP variant, once a block is frozen in place, it becomes invisible. Only the currently controlled block and block-clear animation are visible to the agent. This is known as "Invisibile Tetris" and has a significant human following and some associated human competitions as well.

POMDP Required Capabilities Invisible Tetris is arguably the most difficult task we propose. Standard Tetris is already a fairly hard task, as as far as we know, unsolved by RL. Even Tetris Grand Masters struggle with Invisible Tetris. The arrangement of tiles is complex and constantly shifting. It must be memorized perfectly, and a small error in state quickly compounds as the hidden structure undergoes mutation. This POMDP requires very strong state tracking capabilities and the ability to learn very difficult games.

MDP Recovery from POMDP Each block is visible during the timestep it is locked in place. Considering all observations therefore provides all block positions and recovers the MDP.

H.9 CARTPOLE





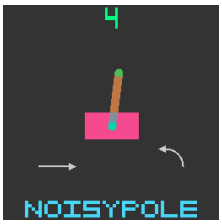
MDP The CartPole environment is a classic control problem framed as a Markov Decision Process (MDP), where the objective is to balance a pole on a moving cart.

POMDP In the POMDP version, after the initial timestep, the cart and pole are hidden from the agent while the arrows representing velocity remain. A horizontal arrow representing the cart's velocity and a curved arrow representing the pole's angular velocity. The size of each arrow is proportional to the magnitude of the corresponding velocity. This creates a pixel-form of the position-masked CartPole problem.

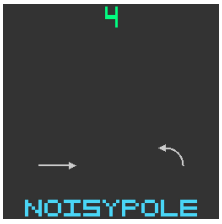
POMDP Required Capabilities To succeed, the agent must integrate velocity signals from all prior timesteps to infer the hidden positional information, directly testing the model's capacity to maintain an internal state from a stream of partial information.

MDP Recovery from POMDP By considering the initial position (rendered) and integrating over all velocities, the agent can predict cart and pole position information.

H.10 NoisyPole



MDP This task is Cartpole (Env. 1) affected by Gaussian noise on position and speed of the cart as well as angle and angular velocity of the pole. The result of noise is still reflected in the arrow magnitudes.



POMDP After the initial timestep, the cart and pole are hidden from the agent, providing only the velocity arrows.

POMDP Required Capabilities This is a harder version of the CartPole POMDP. Morad et al. (2023a) demonstrates that certain memory-free policies can do well in POMDP CartPole, perhaps by learning a time-varying policy. Adding noise results in a better test of memory, as the agent must remember and react to the noise. A simple time-varying policy is insufficient to solve this task.

MDP Recovery from POMDP By considering the initial position (rendered) and integrating over all velocities, the agent can predict cart and pole position information. Noise information is reflected in the velocity observations, allowing integration to recover the full state.

I HARDWARE-ACCELERATED RL

Copying observations from CPU to GPU during rollouts is a major efficiency bottleneck in the training process (Lu et al., 2022). Using JAX, we can implement environments on accelerator hardware, avoiding copy overhead and training policies up to one thousand times faster (Lange, 2022; Koyamada et al., 2023; Matthews et al., 2024b). We highlight Matthews et al. (2024a); Pignatelli et al. (2024); Lu et al. (2024) which offer hardware-accelerated POMDPs. With improved environment throughput, we can consider new training paradigms like Podracer (Hessel et al., 2021; Toledo, 2024). In this work, we focus on the PQN algorithm (Gallici et al., 2024), a simplified podracer version of Q learning. Unlike DQN (Mnih et al., 2015), PQN does not use target networks or replay buffers, and opts for an on-policy TD(λ) objective.

Given the poor sample efficiency of RL, we focus on maintaining high environment throughput. First and foremost, we ensure that all environments are implemented in JAX (Bradbury et al., 2018) in a vectorizable and compilable form to leverage hardware acceleration. Beyond this, we find a number of small tricks can improve performance. Pre-caching sprites and their pixel positions upon reset provides noticeable performance gains, and replacing calls to `jax.lax.cond` with `jax.lax.switch` provides increased throughput.

I.1 VECTORIZED STATE TRANSITIONS AND RENDERING

Our rendering system is built on a canvas framework. We draw to the canvas using primitives, such as letters, numbers, shapes, and more. All rendering functions are pure (without side effects), and may be easily vectorized or compiled. We build our rendering framework on top of `jax.numpy` and does not use any external dependencies. With these tools, anyone can easily render their own custom grid-based or card-based environments for further research and exploration.

Seven of ten of our proposed environments come with fairly intricate rules, requiring multiple iterations of pattern rendering on the canvas. These complex tasks can be broadly split into two rendering categories: grid-based environments, like Battleship, and card-based environments, like CountRecall. Initially, we re-rendered at each step, but this was slow, even with JIT compilation. We found that removing dynamic computations improved performance, which we detail below.

Grid-Based Environments In grid-based games, we divide the entire canvas into a bunch of small square cells, where each cell displays either the same or a unique pattern to reflect the environmental info contained in the current state. We precompute the coordinates of every cell that needs to be drawn on the canvas and stash them in two matrices – one for the x-axis positions and one for the y-axis. During rendering, we leverage `jax.vmap` to handle the process in parallel along the row dimension.

Card-Based Environments We precompute all possible card templates (value, query, and historical cards) during initialization period, avoiding repetitive redrawing of static elements like suits or card positions. This turns dynamic rendering into a fast lookup-and-merge process. For example, when rendering value or query cards, we retrieve pre-drawn templates and apply them to the canvas using masks, skipping pixel-by-pixel logic during runtime. For the history display, we vectorize the drawing of historical cards using `jax.vmap`, generating all variations upfront and later selecting only the visible ones with fast array indexing. By JIT-compiling the render function, we lock these optimizations into a highly efficient, hardware-accelerated pipeline. The benefits are reduced per-frame computation, minimal branching, and GPU-friendly operations—all critical for real-time rendering. Even when rendering complex elements like the history grid, we avoid loops and instead use clever indexing with `jnp.argmax` and masks to overlay the latest valid symbols.

J Related Benchmarks

Below, we list additional relevant benchmarks and their characteristics. A tilde in the MDP Twins columns means that MDP and POMDP twins do not share observation spaces, making causal studies difficult. Our benchmark is the only one with POMDP/MDP twins and GPU acceleration.

Benchmark	POMDP/MDP Twins	GPU
MuJoCo/MJX (Todorov et al., 2012)		✓
Atari/ALE (Bellemare et al., 2013)		
DMLab (Beattie et al., 2016)		
MiniGrid/Navix (Chevalier-Boisvert et al., 2018), (Pignatelli et al., 2024)	$\sim$	
Memory Task Suite (Fortunato et al., 2019)		
MinAtar (Young & Tian, 2019)		
EnvPool Weng et al. (2022)		✓
Gymnax (Lange, 2022)		✓
POMDP Baselines (Ni et al., 2022)	$\sim$	
MemoryGym (Pleines et al., 2022)	✓	
POPGym/POPJaxRL (Morad et al., 2023a), (Lu et al., 2024)	$\sim$	
PGX (Koyamada et al., 2023)		✓
MDP Playground (Rajan et al., 2023)	✓	
Jumanji (Bonnet et al., 2024)		✓
Kinetix (Matthews et al., 2024b)		✓
Craftax (Matthews et al., 2024a)		✓
POPGym Arcade (ours)	✓	✓

K Recurrent Models

In this appendix, we provide detailed descriptions of the recurrent models used in our experiments. We categorize these models into two families: classical recurrences, such as LSTMs and GRUs, which process sequences step-by-step, and associative recurrences, which leverage parallelizable operators for significantly faster computation.

K.1 Classical Recurrences

Classical recurrent models, such as the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), process sequential information in a strictly ordered manner. The computation of the hidden state at any step t, denoted as $h_t = f(h_{t-1}, x_t)$, is fundamentally dependent on the completion of the previous step's computation, h_{t-1}. This creates a sequential chain of operations that cannot be parallelized.

LSTM The LSTM network (Hochreiter & Schmidhuber, 1997) introduces a dedicated cell state c_t to carry information over long sequences, separate from the hidden state h_t. Information flow is regulated by three gates: an input gate i_t, a forget gate f_t, and an output gate o_t. At each timestep t, the LSTM updates its

states as follows:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{9}$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{10}$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{11}$$
$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{12}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{13}$$
$$h_t = o_t \odot \tanh(c_t), \tag{14}$$

where σ is the sigmoid function, $\odot$ denotes element-wise multiplication, and $[h_{t-1}, x_t]$ is the concatenation of the previous hidden state and the current input.

GRU The Gated Recurrent Unit (GRU) (Chung et al., 2014) simplifies the LSTM architecture by merging the cell state and hidden state into a single hidden state vector h_t. It uses two gates: a reset gate r_t and an update gate z_t. The reset gate determines how to combine the new input with the previous hidden state, while the update gate decides how much of the previous hidden state to retain. The update equations are:

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{15}$$
$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{16}$$
$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{17}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{18}$$

The GRU's simpler structure makes it computationally more efficient than the LSTM while often achieving comparable performance.

K.2 Associative Recurrences

Associative recurrent models, also known as linear recurrent models or memoroids (Morad et al., 2024), update the recurrent state with a binary operator $\bullet$ that obeys the associative property

$$h_3 = (x_1 \bullet x_2) \bullet x_3 = x_1 \bullet (x_2 \bullet x_3). \tag{19}$$

Given an associative recurrent update, we can leverage the associative property to rearrange the order of operations. For example, given four inputs x_1, x_2, x_3, x_4, we can compute either

$$h_4 = (((x_1 \bullet x_2) \bullet x_3) \bullet x_4) \qquad h_4 = (x_1 \bullet x_2) \bullet (x_3 \bullet x_4). \tag{20}$$

The former case corresponds to standard recurrent networks, relying on the prior recurrent state to compute the current state, resulting in linear time complexity. In the latter case, we can compute $x_1 \bullet x_2$ and $x_3 \bullet x_4$ in parallel, achieving logarithmic parallel time complexity (Hinze, 2004). Blelloch (1990) provide an associative scan implementation with linear space complexity. Associative recurrences are orders of magnitude faster than classical RNNs in practice, while using much less memory than transformers, making them useful in sample inefficient tasks like RL (Lu et al., 2024).

Our experiments rely on three distinct associative recurrent models: the Fast Autoregressive Transformer (FART) (Katharopoulos et al., 2020), a form of State-Space Model (Gu et al., 2022) called the Linear Recurrent Unit (LRU)(Orvieto et al., 2023), and a GRU (Chung et al., 2014) variant of a Minimal Recurrent Network called the MinGRU (Feng et al., 2024). We provide formal descriptions of each model below.

Linear Transformers Standard transformers have quadratic space complexity from the outer product of keys and queries. The Fast Autoregressive Transformer (FART) (Katharopoulos et al., 2020) replaces softmax attention with a kernelized attention mechanism to achieve linear space complexity and logarithmic

time complexity via associative recurrent updates

$$h_t = h_{t-1} + \phi(W_k x_t)\phi(W_v x_t)^\top \qquad \hat{s}_t = \text{MLP}\left(x + \frac{\phi(W_q x_t)^\top h_t}{\phi(W_q x_t) \cdot \sum_{i=0}^{t} \phi(W_k x_i)}\right). \tag{21}$$

Here, $\phi(x) = 1 + \text{ELU}(x)$ represents a kernel-space projection and the recurrent state h represents the attention matrix. W_k, W_v, W_q represent the key, query, and value projection parameters. To compute $\hat{s}_t$, we multiply the recurrent attention matrix by the query vector and normalize by a scalar.

State-Space Models State-Space Models (SSMs) (Gu et al., 2022) model an associative recurrence via

$$h_t = \overline{W}_A h_{t-1} + \overline{W}_B x_t \qquad \hat{s}_t = \overline{W}_C h_t + \overline{W}_D x_t, \tag{22}$$

where $\overline{W}_A, \overline{W}_B, \overline{W}_C, \overline{W}_D$ are discretizations of carefully initialized trainable parameters W_A, W_B, W_C, W_D. In practice, we initialize W_A, W_B, W_C, W_D deterministically. Deterministic initialization applied to many consecutive SSM layers can result in instabilities, and so Orvieto et al. (2023) proposes a meticulously derived random initialization and new parameterization of SSM parameters. They call their method the Linear Recurrent Unit (LRU).

Minimal Recurrent Networks Feng et al. (2024) revisit the popular Gated Recurrent Unit (GRU) (Chung et al., 2014) and Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) RNNs. They simplify the GRU and LSTM recurrent updates, proposing the MinGRU and MinLSTM with efficient associative recurrent updates. The authors write the MinGRU as

$$h_t = (1 - \sigma(W_1 x_t + b_1)) \odot h_{t-1} + \sigma(W_1 x_t + b_1) \odot \tanh(W_2 x_t + b_2) \qquad \hat{s}_t = h_t, \tag{23}$$

with trainable parameters W_1, b_1, W_2, b_2, sigmoid function σ, and elementwise product $\odot$. The authors find that the MinGRU outperforms multiple SSM variants across offline reinforcement learning tasks, using an offline decision transformer (Chen et al., 2021) framework.

L NETWORK ARCHITECTURE

PQN Our Q-network combines spatial feature extraction with decision policy learning through a JAX/Equinox implementation. The network processes batches of 128×128 RGB images through four convolutional blocks, followed by three dense layers with intermediate normalization (Table 7).

Table 7: Q network architecture

Layer	Parameters	Activation
Conv2D-1	Channels: $3, 64$, Kernel: 5×5, Stride: 2	LeakyReLU
MaxPool2D-1	Pool: 2×2, Stride: 2	–
Conv2D-2	Channels: $64, 128$, Kernel: 3×3, Stride: 2	LeakyReLU
MaxPool2D-2	Pool: 2×2, Stride: 2	–
Conv2D-3	Channels: $128, 256$, Kernel: 3×3, Stride: 2	LeakyReLU
MaxPool2D-3	Pool: 2×2, Stride: 2	–
Conv2D-4	Channels: $256, 512$, Kernel: 1×1, Stride: 1	LeakyReLU
Dense-1	Features: $512, 256$	LeakyReLU
LayerNorm	Shape: 256	–
Dense-2	Features: $256, 256$	LeakyReLU
LayerNorm	Shape: 256	–
Dense-3	Features: $256, 5$	–

PQN $\times$ **RNN** Our recurrent Q-network extends the base architecture with explicit memory handling through a hybrid CNN-RNN-MLP design. The RNN combines a 512-channel input tensor x with a 5-dimensional one-hot encoded last action vector from POPGym Arcade, processes these through its 512-unit hidden state, and generates 256-dimensional output features (Table 8).

Table 8: Recurrent Q network architecture

Layer	Parameters	Activation
Conv2D-1	Channels: $3, 64$, Kernel: 5×5, Stride: 2	LeakyReLU
MaxPool2D-1	Pool: 2×2, Stride: 2	–
Conv2D-2	Channels: $64, 128$, Kernel: 3×3, Stride: 2	LeakyReLU
MaxPool2D-2	Pool: 2×2, Stride: 2	–
Conv2D-3	Channels: $128, 256$, Kernel: 3×3, Stride: 2	LeakyReLU
MaxPool2D-3	Pool: 2×2, Stride: 2	–
Conv2D-4	Channels: $256, 512$, Kernel: 1×1, Stride: 1	LeakyReLU
RNN Cell	Input, Hidden, Output: $517, 512, 256$, Num Layer: 2	
Dense	Features: $256, 5$	–

M EXPERIMENT HYPERPARAMETERS

We used one set of hyperparameters for all our PQN experiments. Please see the following section for the hyperparameter selection methodology.

Table 9: PQN hyperparameters used in all of our experiments. See Gallici et al. (2024) for a detailed description of hyperparameters.

Parameter	**Value**
TOTAL_TIMESTEPS	10e6, 20e6
TOTAL_TIMESTEPS_DECAY	1e6, 2e6
NUM_ENVS	16
NUM_STEPS	128
EPS_START	1.0
EPS_FINISH	0.05
EPS_DECAY	0.25
NUM_MINIBATCHES	16
NUM_EPOCHS	4
NORM_INPUT	False
NORM_TYPE	layer norm
LR	0.00005
MAX_GRAD_NORM	0.5
LR_LINEAR_DECAY	True
REW_SCALE	1.0
GAMMA	0.99
LAMBDA	0.95

N HYPERPARAMETER SELECTION METHODOLOGY

To select hyperparameters, we performed a manual sweep over four model architectures (MLP, MinGRU, LRU, and FART) across six tasks (CartPole, Navigator, BattleShip, MineSweeper, CountRecall, and AutoEncode), and all three difficulty levels. We evaluated the training timesteps, learning rate and schedule, exploration parameters, batch size, number of minibatches per epoch, gradient clipping magnitude. These refer to `TOTAL_TIMESTEPS`, `LR`, `LR_LINEAR_DECAY`, `TOTAL_TIMESTEPS_DECAY`, `EPS_START`, `EPS_FINISH`, `EPS_DECAY`, `NUM_STEPS`, `NUM_ENVS`, `NUM_MINIBATCHES`, `MAX_GRAD_NORM` respectively.

Unlike the original PQN paper which annealed learning epsilon to near-zero over the entire training duration, we found it beneficial to decay more quickly to a slightly higher final epsilon value. When experimenting with epsilon decay to 0.01 and 0.05 over the episode, we noticed that most learning tended to happen near the ends of training. By quickly annealing to 0.05, the policies learned much more quickly. We found annealing epsilon to 0.01 produced suboptimal results.

The batch size is a function of the number of workers and the number of steps taken at each epoch. We found decreasing the number of steps below 128 hurt performance, and increasing it beyond 128 yielded similar results but reduced sample efficiency. With a number of steps at 128, we found that doubling the batch size via double the number of workers, did not produce a noticeable improvement. Performance increased as we increased the number of minibatches to 16, beyond which we did not see meaningful improvements.

We found that changing lambda from the PQN recommended 0.65 to our selected 0.95 resulted in the best returns. We evaluated learning rates including $0.0005, 0.0001, 0.00005, 0.00001$ finally settling on 0.00005. We found linear LR decay outperformed no decay, and that decaying over the first tenth of an episode produced better results than decaying over the full training duration. We tested gradient clipping values of 1.0 and 0.5, selecting the smaller value for training stability despite a minor efficiency reduction.

O HUMAN BASELINES

We added human baseline results in our study. Each human was provided the docstring associated with each game, then played the game using the arrow keys and spacebar. The scores were recorded and uploaded for our analysis.

These baselines are computed from five participants, matched against the five seeds reported in the Fig. 2. We report the per-task breakdown of normalized returns in $[0, 1]$ at the end of this response.

In general, humans tend to perform better on card games, perform similarly on board games, and perform worse on control tasks. Overall, the MLP outperforms humans on MDPs. Humans make up this gap when introducing partial observability, scoring almost the same as the MLP.

Table 10: Return for the human baselines, broken down by task and observability.

Environment	MDP Return	POMDP Return
AutoEncodeEasy	0.79 ± 0.39	0.28 ± 0.09
AutoEncodeMedium	0.82 ± 0.17	0.27 ± 0.08
AutoEncodeHard	0.58 ± 0.12	0.28 ± 0.04
BattleShipEasy	0.96 ± 0.09	0.71 ± 0.08
BattleShipMedium	0.92 ± 0.08	0.73 ± 0.29
BattleShipHard	0.94 ± 0.13	0.66 ± 0.22
CartPoleEasy	0.22 ± 0.11	0.18 ± 0.10
CartPoleMedium	0.16 ± 0.10	0.07 ± 0.05
CartPoleHard	0.12 ± 0.12	0.07 ± 0.05
CountRecallEasy	0.78 ± 0.10	0.35 ± 0.20
CountRecallMedium	0.68 ± 0.17	0.15 ± 0.08
CountRecallHard	0.57 ± 0.11	0.14 ± 0.09
MineSweeperEasy	0.78 ± 0.16	0.79 ± 0.13
MineSweeperMedium	0.41 ± 0.21	0.30 ± 0.14
MineSweeperHard	0.26 ± 0.28	0.17 ± 0.17
NavigatorEasy	0.97 ± 0.02	0.97 ± 0.02
NavigatorMedium	0.97 ± 0.01	0.97 ± 0.01
NavigatorHard	0.97 ± 0.01	0.68 ± 0.67
NoisyCartPoleEasy	0.09 ± 0.05	0.16 ± 0.12
NoisyCartPoleMedium	0.19 ± 0.13	0.17 ± 0.08
NoisyCartPoleHard	0.18 ± 0.09	0.14 ± 0.12
BreakoutEasy	0.55 ± 0.45	0.08 ± 0.03
BreakoutMedium	0.81 ± 0.39	0.06 ± 0.02
BreakoutHard	0.72 ± 0.39	0.06 ± 0.05
SkittlesEasy	0.80 ± 0.33	0.54 ± 0.33
SkittlesMedium	0.76 ± 0.32	0.59 ± 0.39
SkittlesHard	0.69 ± 0.30	0.42 ± 0.24
TetrisEasy	0.21 ± 0.14	0.02 ± 0.03
TetrisMedium	0.18 ± 0.12	0.03 ± 0.03
TetrisHard	0.03 ± 0.04	0.00 ± 0.00

P Compute Resources

We used approximately 231 days equivalent training time on an RTX4090 GPU to produce a little over 2,000 total runs. We used an unknown amount of additional resources to tune hyperparameters and rerun experiments after fixing bugs.

Q LLM Usage

We used LLMs as a writing aid for portions of the paper. We used it to improve grammar and clarity, as well as to restructure the order in which we present our ideas.