Benchmarking Partial Observability in Reinforcement Learning with a Suite of Memory-Improvable Domains

Anonymous authors
Paper under double-blind review

Keywords: reinforcement learning, partial observability, benchmarking

Summary

Mitigating partial observability is a necessary but challenging task for general reinforcement learning algorithms. To improve an algorithm's ability to mitigate partial observability, researchers need comprehensive benchmarks to gauge progress. Most algorithms tackling partial observability are only evaluated on benchmarks with simple forms of state aliasing, such as feature masking and Gaussian noise. Such benchmarks do not represent the many forms of partial observability seen in real domains, like visual occlusion or unknown opponent intent. We argue that a partially observable benchmark should have two key properties. The first is coverage in its forms of partial observability, to ensure an algorithm's generalizability. The second is a large gap between the performance of a agents with more or less state information, all other factors roughly equal. This gap implies that an environment is memory improvable: where performance gains in a domain are from an algorithm's ability to cope with partial observability as opposed to other factors. We introduce best-practice guidelines for empirically benchmarking reinforcement learning under partial observability, as well as the open-source library POBAX: Partially Observable Benchmarks in JAX. We characterize the types of partial observability present in various environments and select representative environments for our benchmark. These environments include localization and mapping, visual control, games, and more. Additionally, we show that these tasks are all memory improvable and require hard-to-learn memory functions, providing a concrete signal for partial observability research. This framework includes recommended hyperparameters as well as algorithm implementations for fast, out-of-the-box evaluation, as well as highly performant environments implemented in JAX for GPU-scalable experimentation.

Contribution(s)

1. We investigate the efficacy of partially observable benchmarks in measuring an algorithm's ability to mitigate partial observability.
   Context: None
2. We introduce the memory improvability property: a partially observable benchmark is memory improvable if there is a gap between agents with more or less state information, all other factors roughly equal.
   Context: None
3. We categorize popular forms of partial observability, and present a list of representative environments that covers these categories.
   Context: This categorization does not cover all forms of partial observability.
4. We present the open-source POBAX benchmark: a suite of memory improvable environments designed to test an algorithm's ability to mitigate partial observability. POBAX is entirely implemented in JAX, allowing for fast and GPU-scalable experimentation.
   Context: None

Benchmarking Partial Observability in Reinforcement Learning with a Suite of Memory-Improvable Domains

Anonymous authors
Paper under double-blind review

Abstract

Mitigating partial observability is a necessary but challenging task for general reinforcement learning algorithms. To improve an algorithm's ability to mitigate partial observability, researchers need comprehensive benchmarks to gauge progress. Most algorithms tackling partial observability are only evaluated on benchmarks with simple forms of state aliasing, such as feature masking and Gaussian noise. Such benchmarks do not represent the many forms of partial observability seen in real domains, like visual occlusion or unknown opponent intent. We argue that a partially observable benchmark should have two key properties. The first is coverage in its forms of partial observability, to ensure an algorithm's generalizability. The second is a large gap between the performance of a agents with more or less state information, all other factors roughly equal. This gap implies that an environment is memory improvable: where performance gains in a domain are from an algorithm's ability to cope with partial observability as opposed to other factors. We introduce best-practice guidelines for empirically benchmarking reinforcement learning under partial observability, as well as the open-source library POBAX: Partially Observable Benchmarks in JAX. We characterize the types of partial observability present in various environments and select representative environments for our benchmark. These environments include localization and mapping, visual control, games, and more. Additionally, we show that these tasks are all memory improvable and require hard-to-learn memory functions, providing a concrete signal for partial observability research. This framework includes recommended hyperparameters as well as algorithm implementations for fast, out-of-the-box evaluation, as well as highly performant environments implemented in JAX for GPU-scalable experimentation.

1 Introduction

Reinforcement learning (Sutton & Barto, 2018) algorithms are being deployed to increasingly complex domains where *partial observability* (Kaelbling et al., 1998) is a fundamental problem. A system is partially observable if its observations contain only partial information about the underlying state. In this setting, agents cannot make effective decisions without reasoning about their past. Resolving partial observability is a necessary but typically challenging task (Zhang et al., 2012), and many system designers try to circumvent this issue with hand-designed environment-specific features (Mnih et al., 2015; Bellemare et al., 2020). The human engineering effort required to resolve partial observability environment by environment reveals the crux of the problem: there exist many different forms of partial observability, each with their own challenges.

To tackle partial observability, researchers develop history summarization algorithms through testing on benchmark partially observable tasks. The classic T-Maze (Bakker, 2001) problem was used to test long-term recall with LSTMs (Hochreiter & Schmidhuber, 1997) in reinforcement learning. The RockSample (Smith & Simmons, 2004) task was originally used to develop partially observable planning algorithms and their capabilities on large state spaces.

Current benchmarks are narrow in their scope of state aliasing, bringing into question whether performance on the benchmark translates to other forms of partial observability. The best-known example is the Atari benchmark (Bellemare et al., 2013), where using only a single frame is partially observable (Hausknecht & Stone, 2015). Similarly, masked continuous control (Han et al., 2020) is a popular benchmark where velocity or positional state information is hidden. Half of the masked continuous control tasks, the agent only requires a few previous time steps to gauge velocity information to recover a Markov state. These benchmarks represent a narrow sampling of partial observability, but constitute a substantial fraction of empirical evaluations (Ni et al., 2022; 2023; Zhao et al., 2023; Lu et al., 2024). Although other benchmarks test on more forms of state aliasing (Morad et al., 2023; Beattie et al., 2016), individual benchmarks lack coverage across the categories of partial observability and often lack justification as to why the selected tasks are good benchmark tasks. In some cases, performance on a partially observable benchmark depends more on implementation details rather than an algorithm's ability to mitigate partial observability (Ni et al., 2022).

Beyond good coverage of the forms of partial observability, a useful benchmark must have a clear signal for evaluating an algorithm's ability to mitigate partial observability. We argue that one such valuable signal is *memory improvability*. An environment is *memory improvable* a gap exists between the performance of agents imbued with more or less state information. This implies that using memory to mitigate partial observability will improve performance in this environment. The performance gap between observations that are partial and those that are (more) complete is exactly the gap that an agent mitigating partial observability ought to close. A large gap indicates that a particular environment can benefit from adding memory; a small or non-existent gap indicates that either the partial observability is not a major issue, or there is some other confounding factor—e.g. featurization scheme, learning dynamics or hyperparameters.

We introduce a new open-source benchmark, POBAX[1]: Partially Observable Benchmarks in JAX. Since testing on all forms of partial observability is untenable, we categorize the different forms of partial observability and select representative environments for our benchmark to ensure that we have coverage of the space of task types. POBAX is a comprehensive suite of new and existing partially observable environments that cover all state aliasing categories of interest described here. These environments include tasks such as localization and mapping, visual control, games and more. Besides requiring hard-to-learn memory, these environments are all memory improvable; as we add more information into the state representation, we see an increase in performance. To show the utility of our benchmark, we test three popular reinforcement learning algorithms designed for mitigating partial observability. We also recommend per-environment hyperparamters for out-of-the-box evaluation of memory learning algorithms. The benchmark is also entirely implemented in JAX (Bradbury et al., 2018), allowing for fast simulation and GPU-scalable experiments.

2 Background and Related Work

We use Markov decision processes (MDPs) (Puterman, 1994) and their extension, partially observable Markov decision processes (POMDPs) (Kaelbling et al., 1998) as the framework for sequential decision making in an unknown environment. An MDP consists of a state space $\mathcal{S}$, action space $\mathcal{A}$, reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, stochastic transition function $T : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$, initial state distribution $p_0 \in \mathcal{S}$, and discount factor $\gamma \in [0, 1]$. The goal of an agent interacting with an MDP is to learn a policy $\pi_{\mathcal{S}} : \mathcal{S} \to \Delta\mathcal{A}$ which tries to maximize its expected discounted returns $V_{\pi_{\mathcal{S}}}(s) = \mathbb{E}_{\pi_{\mathcal{S}}}\left[\sum_{i=0}^{\infty} \gamma^i R_{t+i}\right]$. In the POMDP framework, an agent receives observations $o \in \Omega$ through an observation function $\Phi : \mathcal{S} \to \Delta\Omega$ that maps the underlying hidden states to potentially incomplete state observations. These observations no longer have the Markov property: the observation o_t and action a_t at time step t are no longer a sufficient statistic of history to predict the next

[1] Code: https://anonymous.4open.science/r/pobax-2042

observation and reward, o_{t+1} and r_t, or $Pr(o_{t+1}, r_t \mid o_t, a_t) \neq Pr(o_{t+1}, r_t \mid o_t, a_t, \dots, o_0, a_0)$. Under partial observability, an agent must use its history $h_t := (o_t, a_t, \dots, o_0, a_0) \in \mathcal{H}$ to learn a history-conditioned policy $\pi_\Omega : \mathcal{H} \to \Delta\mathcal{A}$ to maximize returns.

An agent can mitigate partial observability by learning memory functions $\mu : \mathcal{H} \to \mathbb{R}^n$. Memory functions condense past sequences of actions and observations into a memory state $\boldsymbol{m}_t = \mu(h_t)$. Since h_t is variable in size, it is often more efficient and convenient to use recurrent memory functions $\boldsymbol{m}_t = \mu(o_t, a_t, \boldsymbol{m}_{t-1})$. Ideally, a memory function learns to retain information that it needs in future decision making. While traditional approaches have relied on discrete state machines to reason about states (Chrisman, 1992; Peshkin et al., 1999), most modern approaches leverage parameterized deep neural networks (Goodfellow et al., 2016) to learn memory functions. One popular class of neural network memory functions are recurrent neural networks (RNNs) (Amari, 1972; Mozer, 1995), powerful function approximators that can be optimized with truncated backpropagation through time (Jaeger, 2002). Another state-of-the-art class of memory functions are transformers (Vaswani et al., 2017), which is not recurrent, and looks at a fixed context-length window of previous inputs in order to learn memory. For reinforcement learning in partial observability, one can use standard gradient-based reinforcement learning algorithms to learn a neural network memory function capable of summarizing history to mitigate partial observability. The algorithm we use throughout this work for optimization is the popular proximal policy optimization algorithm (PPO) (Schulman et al., 2017). We use this algorithm due to its strong performance in select partially observable environments with RNNs (Ni et al., 2022) and transformers (Ni et al., 2023). We also test on the λ-discrepancy algorithm (Allen et al., 2024), an extension to the recurrent PPO algorithm specifically made for mitigating partial observability.

There have been many forms of benchmark tasks for partial observability. Partially observable tasks were formulated to solve the POMDP planning problem (Zhang et al., 2012), the most well-known instance being the Tiger problem (Kaelbling et al., 1998). In most cases, the scale of these problems are too small and are easily approximated with modern neural networks (Allen et al., 2024). The few exceptions to this rule are benchmarks from POMDP planning algorithms designed to scale up to large state spaces (Silver & Veness, 2010), which we include in our study. Modern deep reinforcement learning algorithms have been tested on a number of difficult and large domains, including single-frame Atari (Hausknecht & Stone, 2015), masked (Han et al., 2020) and visual(Todorov et al., 2012; Ortiz et al., 2024) continuous control, and multiagent systems (Rutherford et al., 2023; Bettini et al., 2024; Lanctot et al., 2019). While there have been benchmarks specifically designed for partial observability (Morad et al., 2023; Osband et al., 2020), these benchmarks tend to have a narrow range of partially observable tasks.

3 Confounding Factors in Assessing Partial Observability Mitigation

The objective of any benchmark is to give researchers a reasonable signal for progress on a class of problems. If the goal of an algorithm is to effectively mitigate partial observability, then progress measured in a benchmark should be from an agent effectively mitigating partial observability, as opposed to other factors. While this may seem obvious, isolating performance increases is a challenging task in practice, considering how many factors affect deep reinforcement learning performance (Henderson et al., 2018). We begin by investigating some potential confounding factors in partially observable reinforcement learning.

There are confounding factors in existing partially observable benchmarks that obfuscate the effects of partial observability. In the Atari benchmark (Bellemare et al., 2013), a single frame is partially observable, whereas four stacked consecutive frames is usually assumed to be fully observable (Mnih et al., 2015). We would expect an agent imbued with state information to outperform an agent that receives only single frames and must do the extra work of resolving partial observability. In reality, results are much more complicated (Hausknecht & Stone, 2015) and different algorithms make gains in different environments. In masked continuous control (Han et al., 2020) one might expect an agent with full state features to perform better than one where certain features are masked

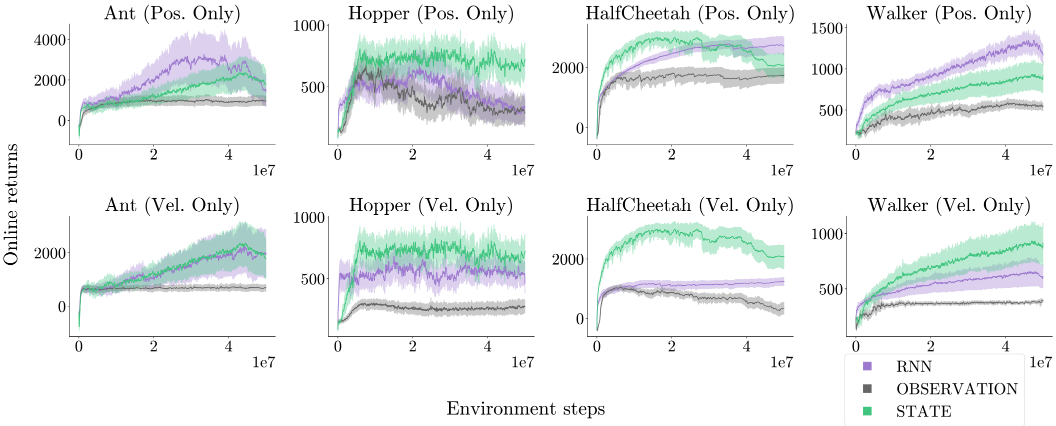


Figure 1: Masked continuous control online undiscounted returns for observations only (gray), full state (green), and an RNN agent (purple) over 30 seeds. Function approximation types play a large role in performance. Full experiment details are presented in Appendix B.4.

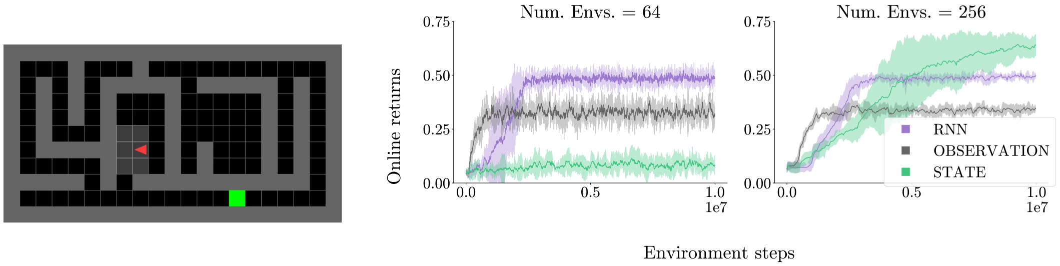


Figure 2: (Left) Image of the DMLab Minigrid maze environment for `maze_id = 01`. (Middle, right) Online discounted returns in this environment comparing performance of using 64 vs 256 parallel environments. Experiments were conducted over 5 seeds.

out. In Figure 1 we show that more often than not, the opposite is true; RNNs under partial observability outperform memoryless agents with fully observable features, as with position-only Ant and Walker. It seems for most of these tasks, agents struggle with other factors besides a lack of information in the state representation.

Other confounding factors such as the choices of hyperparameters or function approximators often impact performance in partial observability benchmarks. An important question to consider is: how much of the improvement is from mitigating partial observability and how much is from other factors? Next, we study the effects of a few important general factors on performance for memory-learning tasks.

3.1 Number of Parallel Environments

Modifying the number of parallel copies of environments can drastically change the performance of a given featurization and algorithm. Reinforcement learning algorithms will use parallel copies of an environment to make uncorrelated minibatches of experience for more stable gradient updates. Figure 2 shows an ablation study on the number of parallel environments in the DeepMind Lab Minigrid domain introduced in Section 6.1. Note that the total number of environment steps used for training remains the same. The difference is in the size of the minibatch for each gradient update. As the number of parallel environments increases, the size of each minibatch increases, but the number of total updates decreases. We generally see improved performance with an increase in the number of parallel environments. Full details of this ablation study are in Appendix B.8. The trade-off for increasing the number of parallel environments is increased memory usage, making experiments less

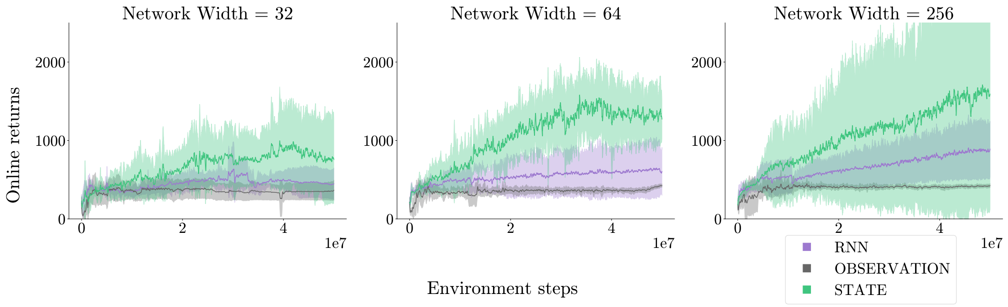


Figure 3: Online undiscounted returns comparing network hidden sizes 32, 64 and 256 (left to right) on velocity-only Walker.

scalable with more parallel environments. To ameliorate this variance, the benchmark we introduce includes recommendations for the number of parallel environments required for each task such that our baseline and skyline agents both learn.

3.2 Network Width

Network width is another general hyperparameter for deep reinforcement learning agents with a sizable but diminishing effect as width increases. The network width is the number of neurons in a neural network's hidden layers, also called its hidden size. In Figure 3 investigate the effect of network width for the velocity-only Walker environment from the masked continuous control benchmark. As network width increases, we see consistent but diminishing improvements in performance. The trade off with increased network width is again a large computational and memory overhead, requiring more resources per experiment. Our benchmark also includes default recommended network widths for each environment. All details of this ablation study are shown in Appendix B.8.

We advocate for choosing general hyperparameter settings for each environment and fixing these settings across all algorithms to ensure a fair comparison between algorithms. Ideally, these settings should also be swept for each algorithm; but with computational resource constraints, sweeping many settings is untenable. As an alternative, our proposed benchmark provides recommended settings for general hyperparameters, including the two studied in this section. This is not to say practitioners should stop sweeping more algorithm-dependent hyperparameters like learning rate. Instead we advocate for a reasonable middle ground for computational feasibility and experimental rigor. Beyond these two hyperparameter settings, many other factors can affect deep reinforcement learning performance. Input featurization and neural network normalization are just a few factors important for performance that we do not investigate in this work. Fixing these confounding factors, we now consider properties that make for a good partially observable benchmark.

4 Memory Improvability

Controlling for confounding factors is not enough to isolate performance gains from mitigating partial observability. We argue that the most important characteristic of an environment is its memory improvability: an indication that performance gains are likely from mitigating partial observability. An environment is memory-improvable if there exists a gap between the performance of agents with less or more state information. If this gap exists, assuming most other factors are equal (e.g. learning algorithm, network size), then gains from a memory-learning algorithm will likely come from mitigating partial observability.

Environments should therefore admit multiple state representations that contain differing amounts of state information such that merely adapting the agent to the new input space is sufficient to achieve a performance improvement with minimal algorithmic changes. Consider the forms of observability in a version of the game *Battleship* (Silver & Veness, 2010) in Figure 4. In this game, players must

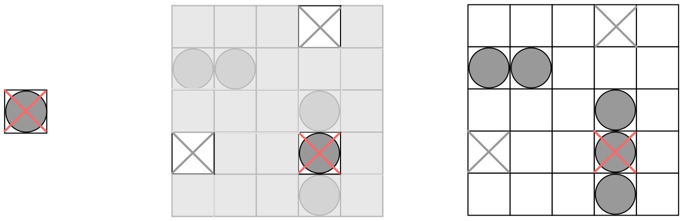


Figure 4: Different levels of observability in 5 × 5 *Battleship*. (Left) Observations in this version of Battleship are whether or not the previous action hit. (Middle) "Perfect memory" observability, where observations include all previous position hit and missed. Grayed out grids are unobservable. (Right) Full observability, where ship positions are also included in observations.

select coordinates on a grid to fire at in order to sink ships. We show three examples of observability here: first is the least observable version, where observations only include whether or not the last shot hit. This poses a particularly hard challenge, since in addition to learning the dynamics of Battleship, the agent must also remember previous shot locations. The second agent has "perfect memory" where the observation is Markov (since no additional information can be gleaned from previous observations) and all previous hits and misses are tabulated in a grid. Lastly, we have the full state observation that also includes ship positions. An agent that learns memory should be able to attain performance matching an agent with perfect memory, whereas optimal performance with full observability is an upper-bound for performance, oftentimes unachievable. Performance with base memoryless observations gives a floor to the performance of an agent, whereas performance with either perfect memory or full state observations gives a ceiling. If a gap exists between the performance of these agents, then an environment is memory improvable. Conversely, it is also possible to create a memory improvability gap by further reducing the amount of information in already-partially-observable state features; for example, features in Battleship that only reveal hits but not misses in Battleship.

With other factors held constant, performance gains by a partial-observability-mitigating algorithm in a memory-improvable environment are more likely due to mitigating partial observability. From Section 3 we know that without memory improvability, performance gains on a partially observable domain could be due to other confounding factors. When the biggest difference between agents is the information in the input features, the gains above the agent with less information are from an agent better mitigating partial observability.

Now that we have described how we intend to evaluate agents with our benchmark, we can assess which environments would make for a good evaluation for mitigating partial observability.

5 Categorizing Partial Observability

To choose representative environments for benchmarking partial observability, we first must define categories of interest that partially observable environments fall into. In the following list, we focus on the different forms that partial observability can take, as opposed to categorization with solution methods in mind. We define eight categories popular in partial observability and example problems for each. Note that environments may fall into multiple categories of partial observability. We emphasize that this is not an exhaustive list of the archetypes of partial observability, but merely popular forms seen throughout reinforcement learning literature.

Noisy state features State features with additive noise. The most popular option for additive noise is to add Gaussian noise to continuous state features: $\phi(\boldsymbol{x}(s)) := \boldsymbol{x}(s) + \delta$, where δ is sampled noise from a multivariate Gaussian with zero mean. Modeling partial observability as additive Gaussian noise is a popular technique in robotics (Thrun et al., 2005). An example of this is noisy Cartpole and Pendulum environments (Morad et al., 2023), where baseline observation-only agents already perform well. Additive state features may not provide the best signal for algorithmic progress in partial observability.

Visual occlusion A portion of the environment's visibility is occluded by other parts of the environment or distance. Visual occlusion is one of the most popular sources of partial observability in both robotics and reinforcement learning, with visual locomotion (Todorov et al., 2012) and occluded maze navigation (Beattie et al., 2016; Chevalier-Boisvert et al., 2023) as popular and challenging existing benchmarks.

Object uncertainty & tracking The state of objects in the environment are unknown, requiring an agent to reason about each object and potentially track it. The classic POMDP benchmark RockSample (Smith & Simmons, 2004) is an apt example, since an agent must test and remember the parity of each rock. Games such as Crafter (Hafner, 2021) contain objects and enemies that may leave the screen which an agent should track or act to observe.

Spatial uncertainty Environments where the agent is required to localize and potentially map its environment. This form of partial observability is a classic task in robotics (Thrun et al., 2005). In reinforcement learning, the aforementioned maze navigation (Beattie et al., 2016) and first-person grid world environments (Chevalier-Boisvert et al., 2023; Pignatelli et al., 2024) are popular examples.

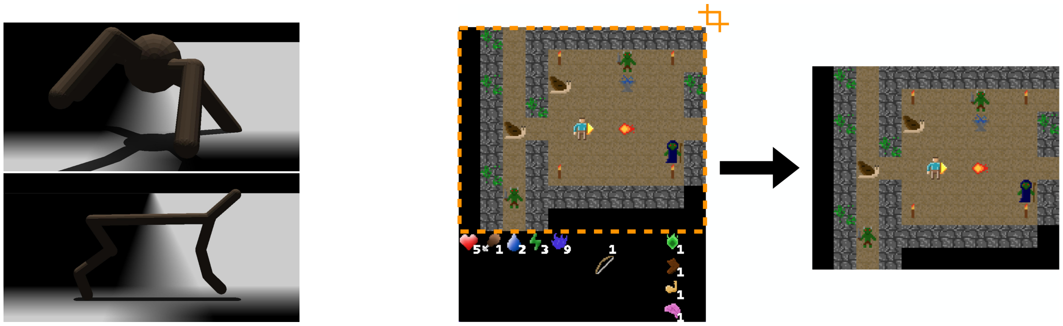
Moment features Environments where state representation is characterized by moments. In continuous control domains (Todorov et al., 2012), position and velocity (first and second moments) of the agent's joints characterize the full state of the system. Environments can be made partially observable by obscuring position or velocity information (Han et al., 2020).

Unknown opposition In multiagent systems, an agent is unaware of the opponent's policy, making the world partially observable. Adding more agents, each with their own policy, exponentially increases the size of the system. Multiagent reinforcement learning is a large field of study with many existing benchmarks (Rutherford et al., 2023; Bettini et al., 2024; Lanctot et al., 2019). Due to the scope of this category, we leave this form of partial observability to these benchmarks.

Episode nonstationarity Tasks where aspects of the environment change over episodes. Maze environments from DeepMind Lab (Beattie et al., 2016) are a classic example of this, where the start and goal positions are randomized at every step for each maze configuration. ProcGen (Cobbe et al., 2019) is an extreme example of this, the environment is partially observable and each episode also instantiates in a randomly generated level of each game.

Needle in a haystack These difficult environments test an agent's ability to memorize a random sequence of events, oftentimes unrelated to one another. An example of this is the diagnostic Autoencode task (Morad et al., 2023), where an agent must repeat back a shuffled deck of 52 cards backwards. In this setting, the only sequence of observations that holds any information about rewards is the sequence of cards shown to the agent—there is no accumulation of information, only a single sequence of actions among exponentially many possibilities of sequences that will result in a reward. We leave out environments of this form because they are diagnostic and purely meant to test memory length, as opposed to partial observability of interest.

Together with memory improvability in Section 4, we are now ready to establish a benchmark for mitigating partial observability.



(a) Visual Mujoco

(b) No-inventory Crafter

Figure 5: Pixel-based environments in POBAX. (Left) Ant and HalfCheetah in visual continuous control. Images are rendered with full JAX support in the Madrona MJX rendering engine (Shacklett, 2024), with the dark coloration due artifacts of the new framework. (Right) Observations in no-inventory Crafter have the agent's inventory cropped out, requiring the agent to remember its items and stats.

6 POBAX: A Fast, Memory-Improvable Benchmark for Reinforcement Learning Under Partial Observability

Partially Observable Benchmarks in JAX (POBAX) is a new suite of reinforcement learning environments for benchmarking partial observability. It includes partially observable environments with hard-to-learn memory functions. These environments cover the categories of partial observability of interest in Section 5, and are all memory improvable with the provided recommended hyperparameter settings. POBAX is also written entirely in JAX (Bradbury et al., 2018) which allows for fast GPU-scalable experimentation.

6.1 Environments

We briefly summarize each environment before testing them on a set of popular reinforcement learning algorithms made for mitigating partial observability. Environment identification strings (for the `get_env` function) are given after their names. Full details of all environments are in Appendix B.

T-Maze (`tmaze_{`$n_{\texttt{length}}$`}`) A small diagnostic benchmark for partial observability and memory length (Bakker, 2001). At the beginning of an episode, the agent is told whether the reward at the end of a hallway is up or down, and the agent must remember this by the time it gets to the T-junction. We recommend using this environment as a sanity check for memory learning algorithms, since the optimal policy's return will always be $4 \times \gamma^{n_{\text{length}}+1}$, where n_{length} is the length of the hallway. Category: *object uncertainty & tracking*

RockSample (`rocksample_11_11` and `rocksample_15_15`) A classic medium-sized problem in POMDP literature (Smith & Simmons, 2004). In *RockSample(11, 11)* and *RockSample(15, 15)*, the agent needs to sample good rocks throughout its environment and exit. Partial observability comes from the need to test each rock with its distance-dependent stochastic sensor. This environment is extendable to the general RockSample(n_{grid}, k) problem, where n_{grid} is the size of the $n_{\text{grid}} \times n_{\text{grid}}$ grid, and k is the number of randomly dispersed rocks in the environment. Category: *object uncertainty*

Battleship (`battleship_10`) Another medium-sized problem based on the board game, also from POMDP planning literature (Silver & Veness, 2010). An agent must hit all 4 ships in a 10×10 grid, and sees only `HIT` or `MISS` at every step. This environment is extendable to any $n_{\text{grid}} \times$

n_{grid} map, with any number of ships of any sizes. Categories: *spatial uncertainty* and *episode nonstationarity*

Masked Mujoco (`Walker-V-v0` and `HalfCheetah-V-v0`) Medium-sized continuous control environments (Walker and HalfCheetah) with only velocity features (Han et al., 2020). In this setting, an agent is required to integrate over its history of velocities to mitigate partial observability. From the experiments in Figure 1, both `Walker-V-v0` and `HalfCheetah-V-v0` are memory improvable and we include them in this benchmark. These environments were made on top of the Brax framework (Freeman et al., 2021). Category: *moment features*

DeepMind Lab MiniGrid mazes (`Navix-DMLab-Maze-{maze_id}-v0`, `maze_id` $\in \{\texttt{01}, \texttt{02}, \texttt{03}\}$) Medium-to-large tasks that are 2D versions of the DeepMind Lab (Beattie et al., 2016) mazes implemented in MiniGrid (Chevalier-Boisvert et al., 2023; Pignatelli et al., 2024), as seen in Figure 2. The agent is randomly initialized to a start position and has to navigate to a randomly sampled goal position. Observations are agent-centric views of the 3×2 area in front of itself, requiring an agent to localize in its environment and find where the goal is. This environment was built on top of the NAVIX framework (Pignatelli et al., 2024). Categories: *spatial uncertainty* and *episode nonstationarity*.

Visual Mujoco (`ant_pixels` and `halfcheetah_pixels`) Large-scale continuous control with single-frame observations (Todorov et al., 2012). An agent is required to gauge its proprioceptive state through frame-by-frame pixel images, as shown in Figure 5a. Using pixel images not only obfuscates the velocity of each joint, but also includes visual occlusion of the other aspects of the state. These environments were built on top of the Brax framework (Freeman et al., 2021). Categories: *visual occlusion* and *moment features*.

No-inventory Crafter (`craftax_pixels`) Large-scale pixel-based alternative version of the Crafter benchmark (Hafner, 2021). In regular Crafter, the agent is already partially observable. To make a memory improvability gap, we make the original state features more partially observable by obscuring the agent's inventory as shown in Figure 5b. This version, called no-inventory Crafter, is memory improvable because there is a performance gap between the original Crafter observations and the no-inventory observations. This environment was built on top of the Craftax framework (Matthews et al., 2024). Categories: *visual occlusion*, *spatial uncertainty*, and *object uncertainty & tracking*.

6.2 Results

We test the above environments on three popular reinforcement learning algorithms designed for mitigating partial observability:

1. Recurrent PPO (Schulman et al., 2017),
2. λ-discrepancy (Allen et al., 2024) with recurrent PPO,
3. Transformer-XL (Parisotto et al., 2020) with PPO.

General hyperparameters for each environment were kept fixed, while algorithm-specific hyperparameters were swept. Both recommended environment hyperparameters and swept-and-selected algorithm hyperparameters are detailed in Appendix A.

To show the utility of our library, we evaluate all three memory-based reinforcement learning algorithms on the POBAX benchmark environments listed in Section 6.1. Results are shown in Figure 6. The gap between observations-only agents (gray) and the additional state information agents (green) imply that the environments are all memory improvable. All three memory-learning algorithms manage to improve upon the performance of the observations-only agent, and underperform the agent with more state information, implying that performance gains are most likely from mitigating partial observability. Results show mean and 95% confidence interval over 30 seeds.

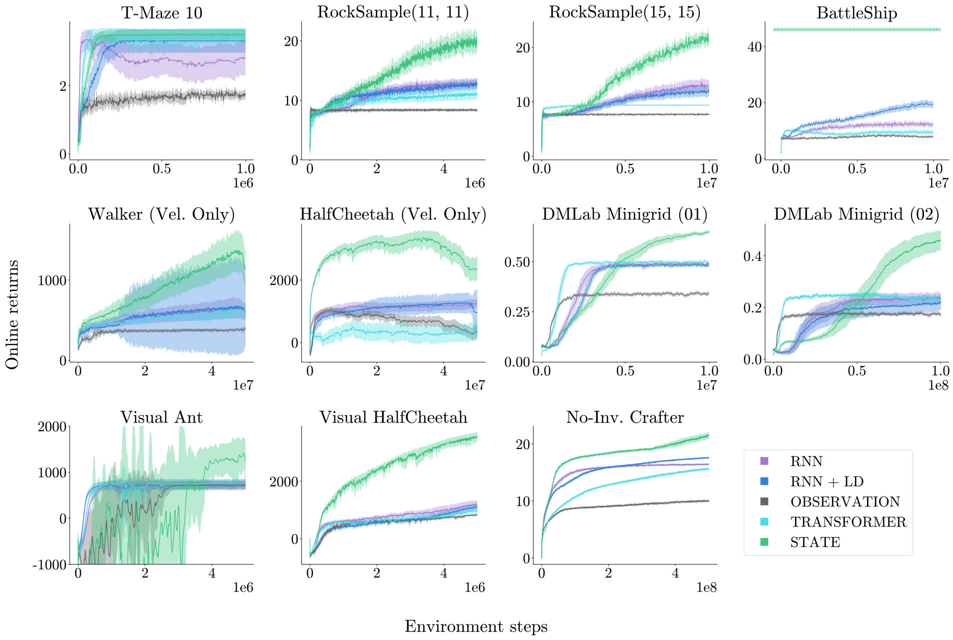


Figure 6: Performance across all POBAX domains. Experiments are run over 30 seeds, with shaded regions denoting a 95% confidence interval.

Ceilings for performance do not have to be the "perfect memory" or fully observable featurizations. In no-inventory Crafter, the "full state" agent is in fact a transformer agent trained on regular Crafter with the inventory included in the observations. The full state agent in BattleShip is the mean performance of an optimal belief policy (Berry, 2011) calculated programmatically. Both of these ceilings represent the mean performance that an algorithm with its original observation feature set should be able to achieve if it can mitigate partial observability effectively.

Finally, the third DMLab Minigrid maze (`maze_id = 03`) was not included in this benchmark due to its difficulty. In addition to requiring complex localization of the environment, these maze environments also pose a hard exploration and sparse reward task for all three algorithms. For `maze_id = 01, 02`, agents were trained on 256 and 512 parallel environments respectively in order for agents to learn effectively. This large number of parallel environments already pose a significant computational overhead, leaving the third task as a difficult, unsolved challenge.

7 Conclusion

Benchmarking an algorithm's ability to mitigate partial observability is challenging due to the scope that partial observability covers and the many confounding factors of deep reinforcement learning. We introduce POBAX: Partially Observable Benchmarks for reinforcement learning in JAX. This open-source benchmark is built around two key properties: coverage over many forms of partial observability and memory improvability. An environment is memory improvable if performance gains are from an algorithm's ability to mitigate partial observability as opposed to other factors. To achieve memory improvability in our benchmark, we investigate the affects of different confounding factors on performance to give a recommended set of hyperparameters for each environment. We then introduce categories of partial observability of interest and select representative environments for our benchmark. Experimental results show that the POBAX benchmark environments are memory improvable, and evaluation of three popular algorithms demonstrate the utility of the benchmark as a signal for research on mitigating partial observability in reinforcement learning.

References

Cameron Allen, Aaron T. Kirtland, Ruo Yu Tao, Sam Lobel, Daniel Scott, Nicholas Petrocelli, Omer Gottesman, Ronald Parr, Michael Littman, and George Konidaris. Mitigating partial observability in decision processes via the lambda discrepancy. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Shun-ichi Amari. Learning patterns and pattern sequences by self-organizing nets of threshold elements. *IEEE Transactions on Computers*, C-21(11):1197–1206, 1972.

Bram Bakker. Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems*, volume 14, 2001.

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. *CoRR*, 2016.

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, jun 2013.

Marc G. Bellemare, Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C. Machado, Subhodeep Moitra, Sameera S. Ponda, and Ziyu Wang. Autonomous Navigation of Stratospheric Balloons using Reinforcement Learning. *Nature*, 588(7836):77–82, 2020.

Nick Berry. Battleship, Dec 2011. URL http://datagenetics.com/blog/december32011/index.html.

Matteo Bettini, Amanda Prorok, and Vincent Moens. Benchmarl: Benchmarking multi-agent reinforcement learning. *Journal of Machine Learning Research*, 25(217):1–10, 2024.

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.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Lonnie Chrisman. Reinforcement learning with perceptual aliasing: the perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pp. 183–188. AAAI Press, 1992.

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Gautier Hamon. transformerXL_PPO_JAX, July 2024. URL https://github.com/Reytuag/transformerXL_PPO_JAX.

Dongqi Han, Kenji Doya, and Jun Tani. Variational recurrent models for solving partially observable control tasks. In *International Conference on Learning Representations*, 2020.

Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable MDPs. In *Proceedings of the 2015 American Association for Artificial Intelligence*, 2015.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'18. AAAI Press, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997. DOI: 10.1162/neco.1997.9.8.1735.

Herbert Jaeger. Tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the echo state network approach. *GMD-Forschungszentrum Informationstechnik, 2002.*, 5, 2002.

Leslie P. Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019.

Chenhao Lu, Ruizhe Shi, Yuyao Liu, Kaizhe Hu, Simon S. Du, and Huazhe Xu. Rethinking transformers in solving pomdps. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

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, 2022.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

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*, 2023.

Michael Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3, 1995.

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*, volume 162, pp. 16691–16723, 2022.

Tianwei Ni, Michel Ma, Benjamin Eysenbach, and Pierre-Luc Bacon. When do transformers shine in RL? decoupling memory from credit assignment. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Joseph Ortiz, Antoine Dedieu, Wolfgang Lehrach, J Swaroop Guntupalli, Carter Wendelken, Ahmad Humayun, Sivaramakrishnan Swaminathan, Guangyao Zhou, Miguel Lazaro-Gredilla, and Kevin Patrick Murphy. DMC-VB: A benchmark for representation learning for control with visual distractors. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado van Hasselt. Behaviour suite for reinforcement learning. In *International Conference on Learning Representations*, 2020.

Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M. Jayakumar, Max Jaderberg, Raphaël Lopez Kaufman, Aidan Clark, Seb Noury, Matthew M. Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR, 2020.

Leonid Peshkin, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning policies with external memory. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pp. 307–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606122.

Eduardo Pignatelli, Jarek Liesen, Robert Tjarko Lange, Chris Lu, Pablo Samuel Castro, and Laura Toni. Navix: Scaling minigrid environments with jax. *arXiv preprint arXiv:2407.19396*, 2024.

Martin L. Puterman. *Markov Decision Processes*. Wiley, 1994.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Gardar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktaschel, Chris Lu, and Jakob Nicolaus Foerster. Jaxmarl: Multi-agent rl environments in jax. *arXiv preprint arXiv:2311.10090*, 2023.

Abraham. Savitzky and M. J. E. Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36(8):1627–1639, 1964.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

Brennan Shacklett. Madrona mjx, Dec 2024. URL https://github.com/shacklettbp/madrona_mjx.

David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems*, 2010.

Trey Smith and Reid Simmons. Heuristic search value iteration for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pp. 520–527, 2004.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, 2018.

Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

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. IEEE, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Zongzhang Zhang, Michael L. Littman, and Xiaoping Chen. Covering number as a complexity measure for POMDP planning and learning. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

Xuanle Zhao, Duzhen Zhang, Liyuan Han, Tielin Zhang, and Bo XU. ODE-based recurrent model-free reinforcement learning for POMDPs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Supplementary Materials

The following content was not necessarily subject to peer review.

A Experimental and Hyperparameter Details

A.1 Algorithms

Our base PPO algorithm is an online learning method designed for training on vectorized environments. It is parallelized using the JAX library (Bradbury et al., 2018) based on a batch experimentation library written in JAX (Lu et al., 2022).

Our experiments consist of two steps. First, we perform a hyperparameter sweep over all environments, using a small number of seeds. Then, we select the best hyperparameters based on the highest area under the curve (AUC) score. After selection, we rerun the best hyperparameters using 30 seeds to generate our results. Note that specific hyperparameters being swept and the number of seeds may vary depending on the domain.

We evaluate four algorithms: Memoryless PPO, Recurrent PPO, λ discrepancy with Recurrent PPO, and Transformer-XL. Memoryless PPO is the standard PPO algorithm without any form of internal memory. This means that it solely relies on the current observation to make decisions. In contrast, the other three algorithms are all memory learning algorithms which incorporate a mechanism to capture past experiences. We use observations-only PPO to show the memory improvability gap in our environments, we run this algorithm twice—once on partial observations and once on full observation to acquire the floor and ceiling of our plots. For Recurrent PPO, λ discrepancy with Recurrent PPO, and Transformer-XL, we also concatenate our action into the observation, which provides additional context to enhance memory learning.

We implement our recurrent PPO model following the approach detailed in (Lu et al., 2022) and we implement the λ-discrepancy algorithm following the implementation of (Allen et al., 2024).

Transformer-XL is a memory-augmented algorithm that extends from the conventional architecture of transformers by incorporating segment-level recurrence. Our algorithm followed the implementation of Hamon (2024). One thing to notice is that traditional transformers use its attention mechanism on a fixed input sequence, during which it will lose temporal information and limit their ability to capture dependencies that span beyond the current window. Transformer-XL overcomes this by storing the hidden states from previous sequence, effectively extending the window of information to allow the agent to acquire information from earlier observations.

A.2 Network Architecture

The general architecture of the network used in all our experiments consist of three parts. First, if the environments have visual inputs, we use either FullImageCNN or SmallImageCNN. Then, we get the feature representations by one of three modules—Memoryless, Recurrent Neural Network (RNN), or Transformer—depending on the algorithms. Finally, we called Actor Critic on the processed features for decision making. The detailed descriptions of the components are provided in the following paragraphs.

Actor Critic All our models use an actor-critic architecture. Both actor and critic networks consist of two layers of standard multi-layer perceptron (MLP) with ReLU activations between layers. There is an additional Categorical or MultivariateNormalDiag functions applied at the end of actor network over actor logits depending on the action space of the environments.

```
ActorCritic(
  Actor(
    Sequential(
      (0): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
      (1): ReLU()
      (2): Dense(in_dims=hidden_size, out_dims=action_dims, bias=True)
      (3): Categorical() or MultivariateNormalDiag()
    )
  )
  Critic(
    Sequential(
      (0): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
      (1): ReLU()
      (2): Dense(in_dims=hidden_size, out_dims=1, bias=True)
    )
  )
)
```

Memoryless The memoryless model is implemented as a four-layer MLP with ReLU activations between layers. The architecture is as follows:

```
Memoryless(
  Sequential(
    (0): Dense(in_dims=input_dim, out_dims=hidden_size, bias=True)
    (1): ReLU()
    (2): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
    (3): ReLU()
    (4): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
    (5): ReLU()
    (6): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
  )
)
```

Recurrent Neural Network Our recurrent neural network consists of a dense layer with ReLU activation, a GRU cell, and another dense layer. In the Battleship environment, we insert an extra dense layer after the first dense layer (which outputs a vector with twice the latent size for this environment). This additional layer processes the first layer's output and the hit-or-miss bit.

```
RNN(
  Sequential(
    (0): Dense(in_dims=input_dim, out_dims=hidden_size, bias=True)
    (1): ReLU()
    (2): GRU(in_dims=hidden_size, hidden_size=hidden_size)
  )
)
BattleshipRNN(
  Sequential(
    (0): Dense(in_dims=input_dim, out_features=2*hidden_size, bias=True)
    (1): ReLU()
    (2): Dense(in_dims=2*hidden_size+1, out_features=hidden_size, bias=True)
    (3): ReLU()
    (3): GRU(input_size=hidden_size, hidden_dim=hidden_size)
  )
)
```

Transformer Our transformer model is taken from a JAX implementation of the transformer in library (Hamon, 2024).

```
Transformer(
  Sequential(
    (0): Encoder(in_dims=input_dim, out_dims=embed_size, bias=True)
    (1): PositionalEmbedding()
    (2): for i in num_layer:
            Transformer(value, query, positional_embedding, mask)
  )
)
```

CNN For environments with visual inputs, we use the following two CNN architectures based on image resolution. For image larger than 20 pixels, we employ a four-layers convolution network defined as follows:

```
FullImageCNN(
  Sequential(
    (0): Conv(features=channels, kernel_size=(7, 7), strides=4)
    (1): ReLU()
    (2): Conv(features=num_channels, kernel_size=(5, 5), strides=2)
    (3): ReLU()
    (4): Conv(features=num_channels, kernel_size=(3, 3), strides=2)
    (5): ReLU()
    (6): Conv(features=num_channels, kernel_size=(3, 3), strides=2)
    (7): Flatten()
    (8): ReLU()
    (9): Dense(in_features=flattened_dim, out_features=hidden_size)
    (10): ReLU()
    (11): Dense(in_features=hidden_size, out_features=hidden_size)
  )
)
```

For image resolution smaller than 20 pixels, we use a three-layers convolutional network with kernel size and strides specific to each domain.

```
SmallImageCNN(
  Sequential(
    (0): Conv(features=num_channels, kernel_size, strides)
    (1): ReLU()
    (2): Conv(features=num_channels, kernel_size, strides)
    (3): ReLU()
    (6): Conv(features=num_channels, kernel_size, strides)
    (7): ReLU()
    (8): Flatten()
    (9): Dense(in_features=flattened_dim, out_features=hidden_size)
  )
)
```

B Environment and Hyperparameter details

All our environments are implemented in JAX (Bradbury et al., 2018) for hardware acceleration. A set of hyperparameters remains constant throughout our experiments. These common settings are provided in Table 1. Unless otherwise specified, these default parameters were used in every experiment. We also note that unless otherwise stated, the "fully observable" agent was trained with a memoryless MLP.

Hyperparam Name	Value	Description
num_envs	4	number of environments run in parallel
default_max_steps_in_episode	1000	maximum steps allowed per episode
num_steps	128	number of steps per update iteration
num_minibatches	4	number of minibatches for gradient updates
double_critic	False	whether to use λ-discrepancy
action_concat	False	whether to concatenate actions with observations
lr	[2.5e-4]	learning rate(s) for the optimizer
lambda0	[0.95]	GAE λ parameter for advantage estimation
lambda1	[0.5]	λ-discrepancy GAE λ parameter
alpha	[1.0]	weighting factor for combining advantages
ld_weight	[0.0]	weight in λ-discrepancy loss
vf_coeff	[0.5]	value coeffient
hidden_size	128	hidden size of network
total_steps	1.5×10^6	total number of training steps
entropy_coeff	0.01	entropy regularization coefficient
clip_eps	0.2	clipping parameter for PPO updates
max_grad_norm	0.5	maximum gradient norm for clipping
anneal_lr	True	whether to anneal the learning rate during training
image_size	32	size of input images
save_checkpoints	False	whether to save checkpoints during training
save_runner_state	False	whether to save the final runner state
seed	2020	base random seed
n_seeds	5	number of seeds to generate from base random seed
qkv_features	256	feature size for transformer query, key, and value
embed_size	256	embedding size used in the transformer model
num_heads	8	number of attention heads in the transformer
num_layers	2	number of transformer layers
window_mem	128	memory window size for caching hidden states
window_grad	64	gradient window size
gating	True	whether to apply gating in transformer
gating_bias	2.0	bias value for the gating mechanism

Table 1: Default Hyperparameter Settings

B.1 T-Maze

T-Maze (Bakker, 2001) is a classic memory testing environment. The agent starts off with equal probability in one of two hallways: a hallway where the reward is up, and a hallway where the reward is down. At the first grid, the agent is informed which hallway its in. After leaving the first grid, the observations no longer inform the agent which hallway it is in, and the agent has to remember its initial observations until it reaches the junction. T-Maze 10 is this maze with a hallway length of 10.

Observation Space The agent's observation is a binary vector with 4 elements. The first two elements dictate which hallway the agent is in (reward up or reward down) and is only set at the start grid. The next element is 1 if the agent is in the hallway. The third elemnt is 1 if the agent is in the junction.

Full Observation Space The full observation space environment has the same observation shape, but the first two elements are always set according to which hallway the agent is in.

Action Space The action space is discrete with 4 possible actions, corresponding to moving in the four cardinal direcitons.

Reward The agent gets +4 for going to the correct side of the junction, and -0.1 for going to the wrong side.

Hyperparameter For T-Maze 10, we conduct a hyperparameter sweep over 5 seeds for all hyperparameters in Table 2 for memoryless, recurrent PPO, Transformer-XL and fully observable. For LD experiments, we sweep through Table 3. We set the hidden size to 32. We train all algorithms for 1×10^6 steps and the best hyperparameters are reported in Table 4. Then we rerun the experiments over 30 seeds using best hyperparameters.

Hyperparameter	
Step size	$\{2.5 \times 10^{-3}, 2.5 \times 10^{-4}, 2.5 \times 10^{-5}, 2.5 \times 10^{-6}\}$
λ_0	$\{0.1, 0.3, 0.5, 0.7, 0.9, 0.95\}$

Table 2: T-Maze-10 hyperparameters swept across non-Lambda discrepancy algorithms.

Hyperparameter	
Step size	$\{2.5 \times 10^{-3}, 2.5 \times 10^{-4}, 2.5 \times 10^{-5}\}$
λ_0	$\{0.1, 0.5, 0.95\}$
λ_1	$\{0.5, 0.7, 0.95\}$
β	$\{0.25, 0.5\}$

Table 3: T-Maze-10 hyperparameters swept across Lambda discrepancy algorithm.

	Step size	$\boldsymbol{\lambda_0}$	$\boldsymbol{\lambda_1}$	$\boldsymbol{\beta}$
Fully Observable	2.5×10^{-4}	0.5	–	–
Memoryless	2.5×10^{-4}	0.3	–	–
RNN	2.5×10^{-3}	0.7	–	–
Transformer-XL	2.5×10^{-4}	0.9	–	–
Lambda Discrepancy	2.5×10^{-4}	0.95	0.95	0.5

Table 4: T-Maze 10 Best Hyperparameters

B.2 Rocksample

Rocksample (Smith & Simmons, 2004) is a navigation problem that simulates a rover searching the environment and assess the rocks. In a rocksample(n, k) problem, n represents the size of the grid and k represents the number of rock in the environments. In our experiments, we consider two variants: Rocksample$(11, 11)$ and Rocksample$(15, 15)$. At the start of each run, rock positions are sampled randomly, and every rock is independently assigned a status of either good or bad. The goal of the agent is to sample all the good rock and avoid all the back ones.

Observation Space The agent's observation is a binary vector with $2n + k$ elements. The first $2n$ elements encode the agent's positions on the board using a two-hot representation. The remaining k elements are only updated after the agent either checks or samples a rock and the corresponding i elements is set to 1 if ith rock appear to be good.

Full Observation Space The full observation space of RockSample is a "perfect memory" state representation, also with $2n + k$ elements. The first $2n$ elements are the same positional encoding. The final k elements keep the most recent observation seen from each k rocks, either from checking or sampling a rock.

Action Space The action space is $(5 + k,)$. The first four dimensions correspond to movement of the agent. The fifth dimension corresponds to sampling a rock in its current position. The last k dimensions correspond to checking each rock. When the agent checks a rock, it receives the rock's correct parity with probability determined by the half-efficiency distance, which is based on the distance from the rock being checked:

$$\frac{1}{2}\left(1 + 2^{-d/max_d}\right), \tag{1}$$

where d is the l_2 distance to the rock, and max_d is the maximum distance from any grid in the domain. This means the closer an agent is to a rock, the more likely the agent will get the correct parity.

Reward The agent gets +10 for exiting to the east. The agent also gets +10 for sampling a good rock, and -10 for sampling a bad rock.

Hyperparameter For both Rocksample(11,11) and Rocksample(15,15), we conduct a hyperparameter sweep over 5 seeds for all hyperparameters in Table 2 for memoryless, recurrent PPO, Transformer-XL and fully observable. For LD experiments, we sweep through Table 3. In Rocksample(11,11), we set the hidden size to 256, the number of environments to 8 and entropy coefficient to 0.2. In Rocksample(15,15), we set the hidden size to 512, number of environments to 16 and entropy coefficient to 0.2. We train all algorithms for 5×10^6 steps and the best hyperparameters are reported in Table 2 and Table 3. Then we rerun the experiments over 30 seeds using best hyperparameters. For both Rocksample(11, 11) and Rocksample(15, 15), the perfect memory agent was trained with an RNN as opposed to a memoryless MLP. This was due to improved function approximation by the RNN, even with a fully observable state.

	Step size	$\boldsymbol{\lambda_0}$	$\boldsymbol{\lambda_1}$	$\boldsymbol{\beta}$
Fully Observable	2.5×10^{-4}	0.5	–	–
Memoryless	2.5×10^{-3}	0.3	–	–
RNN	2.5×10^{-4}	0.95	–	–
Transformer-XL	2.5×10^{-4}	0.1	–	–
Lambda Discrepancy	2.5×10^{-3}	0.1	0.95	0.25

Table 5: Rocksample(11, 11) Best Hyperparameters

	Step size	$\boldsymbol{\lambda_0}$	$\boldsymbol{\lambda_1}$	$\boldsymbol{\beta}$
Fully Observable	2.5×10^{-4}	0.7	–	–
Memoryless	2.5×10^{-3}	0.3	–	–
RNN	2.5×10^{-4}	0.95	–	–
Transformer-XL	2.5×10^{-5}	0.3	–	–
Lambda Discrepancy	2.5×10^{-4}	0.5	0.5	0.25

Table 6: Rocksample(15, 15) Best Hyperparameters

B.3 Battleship

Partially observable battleship (Silver & Veness, 2010) is a less observable variant of the traditional battleship game. The agent has a 10×10 board and four ships with length {5, 4, 3, 2} that are uniformly random generated on the board at the start of an episode. The agent's objective is to hit all parts of each ship under the condition that no position was allowed to hit twice. This setup results in a finite horizon problem, with a maximum of 100 moves (one for each grid position). Therefore, we set the discounted factor $\gamma = 1$. The environment terminates when all positions on the grid with a ship are hit.

Observation Space After each step, the agent only receives a single binary signal. A 0 indicate no ship is hit and a 1 indicate the opposite. To simplify the learning process, we concatenate the agent's last action to the observation. Since the action size is 10×10. The observation space is (101,)

Action Space The action space is defined as $\{1, \ldots, 10\} \times \{1, \ldots, 10\}$, which correspond to row and column number of the board that indicate the next target to hit. Actions are masked at each step to prevent illegal moves.

Reward The agent is penalised -1 for every step it took. When all ships are hit, the agent receive a reward of 100.

Hyperparameter We conducted a hyperparameter sweep over 10 seeds across memoryless, fully observable, RNN, and Transformer-XL models using all the parameters in Table 2, and swept the hyperparameters in Table 3 for LD. All experiments are trained for 1×10^7 steps to select the best hyperparameters. The entropy coefficient was adjusted to 0.05 to encourage exploration, the hidden size was set to 512, and the number of environments was set to 32. Additionally, We set steps-log-frequency to 8 and update-log-frequency to 10. The best hyperparameters selected after the sweep are summarized in Table 7. Then we rerun the experiments over 30 seeds using best hyperparameters.

	Step size	λ_0	λ_1	β
Fully Observable	–	–	–	–
Memoryless	2.5×10^{-3}	0.1	–	–
RNN	2.5×10^{-3}	0.7	–	–
Transformer-XL	2.5×10^{-5}	0.1	–	–
Lambda Discrepancy	2.5×10^{-3}	0.1	0.95	0.5

Table 7: Battleship Best Hyperparameters

B.4 Masked Continuous Control

Masked continuous control are Mujoco environments (Todorov et al., 2012; Freeman et al., 2021) with only velocity (Vel. Only) or only positional (Pos. Only) features.

Observation Space The observation space for each environment changes depending on which environment is used and what variables are masked. We refer to our code repository (`https://anonymous.4open.science/r/pobax-2042`) for full details of each observation space, as well as the Brax documentation (Freeman et al., 2021) for details of the original observation space. Note that all masked continuous control results presented in this work was smoothed using a Savitzky-Golay filter (Savitzky & Golay, 1964) with a window of 30 and a polynomial degree of 3.

Fully Observable Observation Space The full observation of each environment are equivalent to the full observations in each Brax environment.

Reward and Action Space The reward and action space are similar to the corresponding Brax environment.

Hyperparameter For all environments, we conduct a hyperparam sweep over 5 seeds for all hyperparameters in Table 2, Table 3. We trained for 5×10^7 steps. The hidden size is set to 256, step-log-frequency to 16, update-log-frequency to 20. For transformer, the embed size is set to 96. We list the best hyperparameters for the Walker-V and HalfCheetah-V environments in Tables 8 and 9 as they appear in our benchmark. We refer to our codebase for the best hyperparameters selected for the full masked mujoco hyperparameter sweep. For both of these environments, we use RNN function approximation for the fully observable results, due to better performance.

Table 8: Halfcheetah-V Best Hyperparameters

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.1	–	–
Memoryless	2.5×10^{-4}	0.7	–	–
RNN	2.5×10^{-4}	0.9	–	–
Transformer-XL	2.5×10^{-4}	0.9	–	–
Lambda Discrepancy	2.5×10^{-5}	0.95	0.7	0.25

Table 9: Walker-V Best Hyperparameters

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.9	–	–
Memoryless	2.5×10^{-4}	0.95	–	–
RNN	2.5×10^{-4}	0.95	–	–
Transformer-XL	2.5×10^{-5}	0.95	–	–
Lambda Discrepancy	2.5×10^{-4}	0.95	0.95	0.5

B.5 DeepMind Lab MiniGrid mazes

DeepMind Lab MiniGrid mazes are MiniGrid (Chevalier-Boisvert et al., 2023; Pignatelli et al., 2024) mazes with the maze layouts from the DeepMind Lab (Beattie et al., 2016) "navigation levels with a static map layout" as shown in Figure 7. These three mazes get increasingly complex and large. At the beginning of every episode, both agent start state and goal state are randomly initialized. Maximum number of episode steps is 2000, 4000 and 6000 for each maze, from lowest ID to highest ID.

Observation Space One-hot first-person images of size $(2, 3, 2)$, where the two channels represent the wall positions and goal locations in the 2×3 grids in front of the agent.

Fully Observable Observation Space Agent-centric one-hot images of size $(2h-1, 2w-1, 2+4)$, where h and w are the height and width of each maze. Position is encoded by shifting the map so that the agent is always in the center. The first two channels represent the walls and goal positions. The last four dimensions represent a one-hot encoding (across the channels) of the direction the agent is facing.

Action Space Discrete space of 3 actions, representing `forward`, `turn left` and `turn right`.

Reward The agent gets $+1$ once it reaches the goal, with a discount factor of $\gamma = 0.99$.

Hyperparameter For both Navix-01 and Navix-02, we conducted our experiments over 5 seeds for all hyperparameters in Table 10, 11. The hidden size is set to 512 and the embed size for transformer experiment is set to 220. The number of environment is set to 256 in Navix-01 and 512 in Navix-02. Navix-01 is trained for 1×10^7 steps and Navix-02 is trained for 1×10^8 steps. The best hyperparameters are provided in Table 12, 13.

Hyperparameter	
Step size	$\{2.5 \times 10^{-3}, 2.5 \times 10^{-4}, 2.5 \times 10^{-5}, 2.5 \times 10^{-6}\}$
λ_0	$\{0.1, 0.5, 0.7, 0.9, 0.95\}$

Table 10: DeepMind Lab MiniGrid Maze hyperparameters swept across non-Lambda discrepancy algorithms.

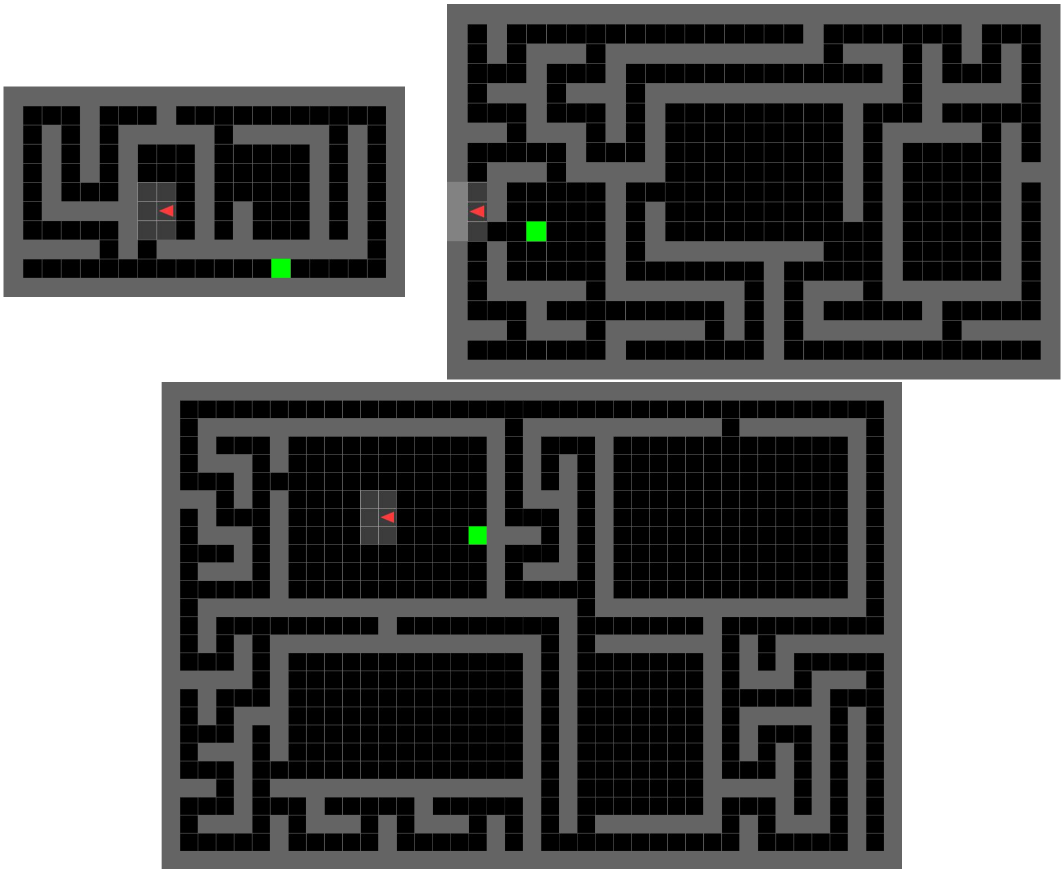


Figure 7: (Left to right, top to down) Three DeepMind Lab MiniGrid mazes, `maze_id = 01, 02, 03`. As `maze_id` increases, maze complexity and size increases as well.

Hyperparameter	
Step size	$\{2.5 \times 10^{-4}, 2.5 \times 10^{-5}\}$
λ_0	$\{0.1, 0.95\}$
λ_1	$\{0.5, 0.7, 0.95\}$
β	$\{0.25, 0.5\}$

Table 11: DeepMind Lab MiniGrid Maze hyperparameters swept across Lambda discrepancy algorithm.

B.6 Visual Continuous Control

Visual continuous control are Mujoco environments with pixel features. We integrate the Madrona MJX (Shacklett, 2024) renderer on top of Brax environments to enable just-in-time (JIT) compilation over rendering in JAX. Note that the Madrona MJX renderer supports only a single batched environment. Thus, we remove the parallelization of training hyperparameters in our algorithms specifically for visual mujoco experiment. Also note that all visual continuous control results presented in this work were also smoothed with a Savitzky-Golay filter (Savitzky & Golay, 1964) with a window of 30 and a polynomial degree of 3.

Observation Space To ensure that the environment is memory-improvable, we do not use frame stacking. The observation space is a single frame represent the current view of the agent. Height and width of the image are determined by the image size hyperparameter. In our experiments, we set the image size to 32, so the observation is (32, 32, 3) for visual mujoco experiments.

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.95	–	–
Memoryless	2.5×10^{-4}	0.95	–	–
RNN	2.5×10^{-4}	0.9	–	–
Transformer-XL	2.5×10^{-4}	0.95	–	–
Lambda Discrepancy	2.5×10^{-4}	0.95	0.5	0.25

Table 12: DeepMind Lab MiniGrid Maze Level 1 Best Hyperparameters

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.95	–	–
Memoryless	2.5×10^{-3}	0.95	–	–
RNN	2.5×10^{-4}	0.95	–	–
Transformer-XL	2.5×10^{-4}	0.95	–	–
Lambda Discrepancy	2.5×10^{-4}	0.95	0.95	0.25

Table 13: DeepMind Lab MiniGrid Maze Level 2 Best Hyperparameters

Fully Observable Observation Space For fully observable ceiling, we use the observation space from original Brax (Freeman et al., 2021) environments for the HalfCheetah and Ant environments. Halfcheetah has observation space (18,) and Ant has observation space (27,).

Action Space We use Brax HalfCheetah and Ant action space to evaluate all our algorithm. HalfCheetah has a continuous action space of shape (6,) and Ant has a continuous action space of shape (8,). The values of actions in both of the environments fall between -1 and 1, where each component representing the torque applied to a specific part of the agent.

Reward The reward function of Ant consists three parts. The agent is rewarded for every second it survives and It is also rewarded for moving in the desired direction. It is penalised for taking too large action and also if the external force is too large. The reward function of Halfcheetah has two parts. The agent is rewarded for going in forward direction and it is penalised for taking too large action.

Hyperparameter We swept both Halfcheetah and Ant over 3 seeds for all hyperparameter in Table 10, 11 and train for 5×10^6 to get the best hyperparameters. Specifically, we set the hidden size to 512. For transformer experiments, we set the embed size to 220 to match the total number of parameters in recurrent PPO. The rest hyperparameters are default. We present the best hyperparameters found for environments in Table 14, 15. After the selection, we rerun the experiments over 30 seeds using best hyperparameters.

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.5	–	–
Memoryless	2.5×10^{-4}	0.1	–	–
RNN	2.5×10^{-4}	0.7	–	–
Transformer-XL	2.5×10^{-4}	0.5	–	–
Lambda Discrepancy	2.5×10^{-4}	0.1	0.5	0.5

Table 14: Ant Best Hyperparameters

B.7 No-inventory Crafter

No-inventory Crafter is a more partially observable variant of Crafter (Hafner, 2021). This environments was built on top of the Craftax framework (Matthews et al., 2024). Craftax is a version of

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.5	–	–
Memoryless	2.5×10^{-4}	0.7	–	–
RNN	2.5×10^{-4}	0.7	–	–
Transformer-XL	2.5×10^{-4}	0.7	–	–
Lambda Discrepancy	2.5×10^{-4}	0.95	0.95	0.5

Table 15: Halfcheetah Best Hyperparameters

Crafter that is implemented in JAX (Bradbury et al., 2018). On top of their work, we furthur made this environment more partially observable by masking the inventory located at the bottom of an observation.

Observation Space The original Craftax observation consists of a grid of 13 by 9 pixel squares, where each square is 10×10 pixels, making the original observation $(130, 90, 3)$. To make it more efficient, our No-inventory Crafter pixel observation has the same form as Craftax, but we downscaled the square from 10×10 to 3×3 and then we mask the pixels that correspond to the inventory, resulting in a final observation shape $(27, 33, 3)$.

Fully Observable Observation Space For our fully observable ceiling, we use the Craftax symbolic observation, which has shape (8268,). The first section is the flattened map representation containing information about block, item, mob and light level. Then the next section is the inventory, followed by potions, player's intrinsics, player's direction, armour and special values.

Action Space We use the same action space with Craftax, which is a discrete action space of 43. Note that every action can be taken at any time, thus attempting to execute an action that is not available will result in a no-op action.

Reward We adopt the same reward scheme used in Craftax. The agent receive the reward the first time it complete an achievement. There are a total 65 achievements which are characterized into 4 categories: 'Basic', 'Intermediate', 'Advanced', and 'Very Advanced', for which the agent is rewarded 1, 3, 5, 8 points respectively. The agent is also penalised 0.1 point every point of damage it took and rewarded 0.1 every health it recovered.

Hyperparameter We swept Craftax over 3 seeds for all hyperparameters in Table 10 and 11 and train for 5×10^8 steps across all the algorithms. We set the number of environments to 256 and hidden size to 512. For the transformer experiments, we set the embed size to 220 to match the total number of parameters in recurrent PPO. The best hyperparameters selected after the sweep are summarized in Table 16. After selection, we rerun the experiments over 30 seeds with the best hyperparameters.

Table 16: No-inventory Crafter Best Hyperparameters

	Step size	λ_0	λ_1	β
Fully Observable	2.5×10^{-4}	0.7	–	–
Memoryless	2.5×10^{-5}	0.95	–	–
RNN	2.5×10^{-4}	0.5	–	–
Transformer-XL	2.5×10^{-5}	0.7	–	–
Lambda Discrepancy	2.5×10^{-4}	0.1	0.95	0.25

B.8 Ablation studies

Here we describe details of the number of parallel environments and network width ablation studies. Both studies were conducted over 5 seeds. Hyperparameters for the ablation study on number of parallel environments swept were the same as Appendix B.5 for `maze_id = 01`, except with the additional sweep of `num_envs`$\in$ $(64, 256)$. Hyperparameters for the ablation study on network width were the same as Appendix B.4, except with the additional sweep of `hidden_size`$\in$ $(32, 64, 256)$. Best performance was taken over discounted returns.