Neural Value Iteration

Yang You[1] **Ufuk Çakır**[2] **Alex Schutz**[2] **Nick Hawes**[2]

[1]United Kingdom Atomic Energy Authority,
[2]Oxford Robotics Institute, University of Oxford,

Abstract

The value function of a POMDP exhibits the piecewise-linear-convex (PWLC) property and can be represented as a finite set of hyperplanes, known as α-vectors. Most state-of-the-art POMDP solvers (offline planners) follow the point-based value iteration scheme, which performs Bellman backups on α-vectors at reachable belief points until convergence. However, since each α-vector is $|S|$-dimensional, these methods quickly become intractable for large-scale problems due to the prohibitive computational cost of Bellman backups. In this work, we demonstrate that the PWLC representation of POMDP value functions can be generalized by replacing the alpha-vector sets with a finite set of neural networks. This insight enables a novel POMDP planning algorithm, called *Neural Value Iteration*, which combines the generalization capability of neural networks with the classical value iteration framework. Requiring only a black-box simulator, our approach scales to extremely large POMDPs that are intractable for previous offline planning methods.

1 INTRODUCTION

The partially observable Markov decision process (POMDP) framework models decision-making problems where the true state is hidden and transitions are stochastic. In a POMDP, the agent maintains a *belief* (a probability distribution over states) and seeks an optimal policy mapping beliefs to actions to maximize expected return. However, solving POMDPs optimally is computationally intractable: PSPACE-complete in the finite-horizon case [Papadimitriou and Tsitsiklis, 1987] and undecidable for infinite horizons [Madani et al., 1999].

It has been proved that the POMDP's optimal value function V_t^* at each time step t is piecewise linear and convex (PWLC) [Sondik, 1971]. As a result, V_t^* can be efficiently represented by a finite set of $|S|$-dimensional hyperplanes, known as α-vectors, where S denotes the state space. By leveraging this PWLC structure, value iteration methods for POMDPs (e.g., dynamic programming updates) can be performed by iteratively maintaining and updating a finite set of α-vectors [Pineau et al., 2003, Smith and Simmons, 2004, 2005, Kurniawati et al., 2008, Bai et al., 2010]. However, since each α-vector spans the entire state space, these methods are difficult to scale due to memory and computation costs. Even efficient solvers like SARSOP struggle with domains containing millions of states, where optimizing large sets of α-vectors becomes intractable. To improve scalability in continuous-state POMDPs, MCVI [Bai et al., 2010] implicitly represents α-vectors using nodes in a policy graph (i.e., a finite-state controller) and performs approximate Bellman backups via Monte Carlo (MC) simulations. Despite its scalability advantages, MCVI suffers from performance degradation in large problems with long horizons due to the high cost of MC simulations, limiting its practicality unless techniques like macro-actions [Lim et al., 2011] are used to reduce planning depth and computational overhead.

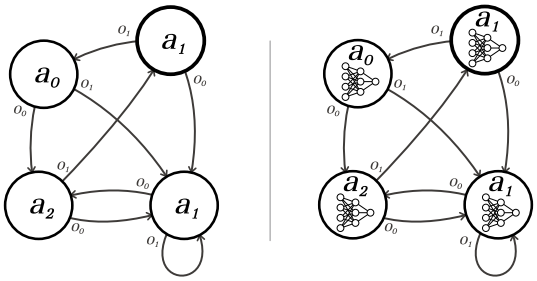


Figure 1: A finite-state controller (FSC) vs. a finite-network controller (FNC). Each FNC node stores a neural network that explicitly approximates an α-vector.

In this paper, based on the classic value iteration (VI) scheme, we aim to further extend its scalability to solve

Proceedings of the 42nd Conference on Uncertainty in Artificial Intelligence (UAI 2026), PMLR 337:7896–7912.

very large POMDPs. To do so, we address a fundamental question: *Is there a better way to represent a value function that is more suitable for such large-state POMDPs?* Motivated by the answer to this question, we propose a new perspective: one can leverage the PWLC property and efficiently approximate a POMDP's value function using a finite set of neural networks. This insight leads to a new policy representation, the *finite-network controller* (FNC), where each node holds a neural network that explicitly models an α-vector (fig. 1), allowing compact policies even in complex domains. We then introduce *Neural Value Iteration* (NVI), a new offline solver that performs value iteration directly over these networks. To our knowledge, NVI is the first offline POMDP solver to scale to problems as large as RockSample(20, 20), with over 400 million states—far beyond the capability of existing general offline POMDP solvers.

2 RELATED WORK

POMDP planning methods are broadly categorized as offline or online. Offline solvers compute a complete policy prior to execution, typically in a compact form for efficient runtime deployment. Online planners instead interleave planning and execution, selecting actions at each step via lookahead search or sampling. Here, we focus on offline approaches most relevant to our work and refer readers to comprehensive surveys for broader coverage [Lauri et al., 2022, Kurniawati, 2022].

Most offline solvers [Smith and Simmons, 2004, 2005, Kurniawati et al., 2008] iteratively update the value function using Bellman backups over a finite set of α-vectors. These methods and their variants [Bai et al., 2010, Lim et al., 2011] follow a *bottom-up* strategy, constructing the optimal value function backward in time from horizon h to 1. Modern algorithms such as HSVI and SARSOP improve efficiency by restricting backups to representative beliefs selected via reachability heuristics, substantially accelerating convergence. Nevertheless, for very large POMDPs, even these approaches remain computationally intractable. An exception among recent offline methods is POMCGS [You et al., 2025], which doesn't follow a value iteration and instead uses a *top-down* strategy inspired by online MCTS-based planners [Silver and Veness, 2010, Somani et al., 2013, Sunberg and Kochenderfer, 2017, Wu et al., 2021, Kocsis and Szepesvári, 2006]. It treats the POMDP as a belief-MDP and performs tree search directly in belief space, merging similar nodes to compact the policy structure and reduce redundant computation.

Several techniques have been explored to scale POMDP planning to large-scale or continuous domains. For example, Algebraic Decision Diagrams (ADDs) have been used to represent value functions in factored POMDPs with known models Shani et al. [2008], Sim et al. [2008]. These approaches exploit the logical structure of discrete factored state variables to reduce memory requirements, but still rely on exact model-based Bellman backups. For general POMDPs, another common approach is to approximate beliefs or value functions using parametric representations, such as Gaussian mixtures [Porta et al., 2006]. However, these representations often oversmooth discontinuities (e.g., walls or obstacles) and may require many components for accurate approximation, limiting scalability. Moreover, [Porta et al., 2006] relies on explicit parametric assumptions such as linear-Gaussian transitions and Gaussian-mixture observations, restricting applicability to general POMDPs. In reinforcement learning, POMDPs are typically addressed by learning belief surrogates through recurrent neural networks [Hausknecht and Stone, 2015, Schulman et al., 2017, Karkus et al., 2017], with value functions defined over latent states. Recent approaches such as BetaZero [Moss et al., 2024] approximate value functions using simplified belief statistics (e.g., mean and variance) as heuristics within MCTS planning. However, none of these methods explicitly exploit the PWLC structure of POMDP value functions, limiting their alignment with value iteration theory.

In this work, we offer a new perspective: the PWLC structure of the optimal value function can be preserved by representing each α-vector with a neural network. This insight leads to the proposed *finite-network controller* (FNC) and the corresponding offline algorithm, *Neural Value Iteration* (NVI), which enables scalable planning for general POMDPs with large state spaces, without requiring specific structural assumptions or access to explicit transition and observation models.

3 BACKGROUND

3.1 POMDPS

A POMDP defines a single-agent decision-making problem in an environment with uncertain dynamics where the underlying states cannot be directly observed. A POMDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \Omega, T, O, r, b_0, \gamma \rangle$ where (i) $\mathcal{S}$ is the set of states; (ii) $\mathcal{A}$ is the set of actions; (iii) Ω is the set of observations; (iv) $T(s', s, a) = \Pr(s'|s, a)$ is the transition function, which encodes the probability of reaching the next state s' given the current state s and an executed action a); (v) $O(o, s', a) = \Pr(o|s', a)$ is the observation function; it indicates the probability of receiving an observation o given the next state s' reached while performing action a; (vi) $r(s, a)$ is the reward function, which gives the instant reward when performing action a at state s; (vii) b_0 is an initial probability distribution over states; (viii) γ is the discount factor.

In a POMDP, the agent maintains a belief b at each time step which represents the probability distribution of possible current states. When the agent executes an action a and

receives a new observation o, the agent updates its belief as $b^{a,o} = \tau(b, a, o)$, where

$$b^{a,o}(s') = \frac{O(o, s', a) \sum_{s \in S} T(s', s, a) b(s)}{\sum_{s'' \in S} \left[O(o, s'', a) \sum_{s \in S} T(s', s, a) b(s) \right]} \tag{1}$$

To simplify the expression, we use $\tau(b, a, o)$ to refer to the belief update process for $b^{a,o}$ in following sections. The goal of a POMDP is to compute a policy π that maps beliefs to agent actions that maximize the expected discounted accumulated reward.

3.2 ALPHA VECTORS AND FINITE-STATE CONTROLLERS

A POMDP value function V can be represented by a finite set of α-vectors, $\Gamma = \{\alpha_1, \ldots, \alpha_n\}$, where each α is associated with an action. Given a belief b, the value is computed as:

$$V(b) = \max_{\alpha \in \Gamma} \sum_{s \in S} b(s)\, \alpha(s), \tag{2}$$

and the optimal action is the one associated with the maximizing α-vector.

It is well known that such a value function can be equivalently encoded as a finite-state controller (FSC), or policy graph, where each node corresponds to an α-vector and is labeled with an action and a transition rule based on observations [Hansen, 1997]. This transformation yields compact policy representations and efficient execution via table lookups.

Definition 1 *A (deterministic) Finite-State Controller for a POMDP is a tuple $fsc = \langle N, n_0, \psi, \eta \rangle$, where: 1. N is a finite set of controller nodes (machine states), 2. $n_0 \in N$ is the initial node, 3. $\psi : N \to A$ maps each node to an action in A, 4. $\eta : N \times \Omega \to N$ maps a node and observation $o \in \Omega$ to the next node. At each step, the FSC selects an action via ψ and updates its node using η, defining a reactive policy over partial observations.*

3.3 BELLMAN BACKUPS IN POMDPS

In this section, we describe the core Bellman backup operation that underpins our contribution. For a broader overview of point-based methods, we refer the reader to [Shani et al., 2013]. A POMDP's optimal value function can be computed through value iterations, and each iteration is commonly referred to as a *Bellman Backup*. This backup process updates a new value function V' from the current value function V as:

Algorithm 1: Bellman Backup

Function `Backup` (Γ, b) **:**
  foreach $a \in A$ ***and*** $o \in \Omega$ **do**
    $\alpha_{a,o} \leftarrow \arg\max_{\alpha \in \Gamma} (\alpha \cdot \tau(b, a, o))$
  foreach $a \in A$ ***and*** $s \in S$ **do**
    $\alpha_a(s) \leftarrow R(s, a) + \gamma \sum_{o \in \Omega} \sum_{s' \in S} T(s, a, s')\, O(s', a, o)\, \alpha_{a,o}(s')$
  $\alpha' \leftarrow \arg\max_{a \in A} (\alpha_a \cdot b)$;
  return $\Gamma \cup \{\alpha'\}$

$$V'(b) = \max_{a \in A} \left[R(b, a) + \gamma \sum_{o \in \Omega} \Pr(o|b, a) V(b^{a,o}) \right] \tag{3}$$

$$= \max_{a \in A} \left[R(b, a) + \gamma \sum_{o \in \Omega} \Pr(o|b, a) \max_{\alpha \in \Gamma} \sum_{s' \in S} b^{a,o}(s') \alpha(s') \right]. \tag{4}$$

The complexity of a raw backup computation in Equation (3) is $O(|V| \times |A| \times |\Omega| \times |S|^2 + |A| \times |S| \times |V|^{\Omega})$, which means this kind of naive value iteration can only be done in very small problems with very few horizons.

One of the key breakthroughs in offline POMDP planning is the development of *point-based value iteration* (PBVI) methods, which demonstrate that full Bellman backups over the entire belief space, as required by the raw value iteration in Equation (3), are unnecessary. Instead, PBVI methods maintain a finite set of representative beliefs $\mathcal{B}$ and reuse cached α-vectors for efficient computation as shown in Alg. 1. This compact backup operation generates a new α-vector α' for a given belief b, and add it to the α-vector set $\Gamma \leftarrow \Gamma \cup \{\alpha'\}$, which represents the updated value function V'.

Nonetheless, when the state space is large or continuous, even this backup process becomes intractable due to the need to sum over all possible states. To address this challenge, Monte Carlo Value Iteration (MCVI) [Bai et al., 2010] was proposed. MCVI avoids explicit loops over the entire state space by sampling states and implicitly representing α-vectors using policy graph nodes. It approximates the Bellman backup using Monte Carlo simulations, a procedure referred to as *MC-Backup*. However, the lack of explicit α-vector representations in MC-Backup prevents caching and reusing computation across belief points. As a result, MCVI often requires a large amount of simulation to achieve competitive performance, which limits its solution quality in large-scale problems under constrained computational budgets.

4 NEURAL VALUE ITERATION

This section presents our main contribution. At a high level, our algorithm maintains and trains a finite set of neural networks during planning to approximate the POMDP value function over a set of representative beliefs. The training data for each network comes from sampling outcomes.

4.1 REPRESENTING V WITH A FINITE SET OF NEURAL NETWORKS

A common approach is to approximate the POMDP value function as a parametric mapping from the belief space $\mathbb{B}$ to $\mathbb{R}$. However, this typically requires simplified belief representations (e.g., means and variances [Moss et al., 2024] or RNN hidden states [Hausknecht and Stone, 2015, Schulman et al., 2017]).

In this paper, we propose to revisit the key PWLC structure exploited by previous value iteration methods, which allows a POMDP value function V to be represented as the maximum over a finite set of α-vectors, i.e., $V(b) = \max_{\alpha \in \Gamma} b \cdot \alpha$. We then ask a fundamental question: what is an α-vector mathematically? A simple but important answer is that each α-vector is a mapping from $\mathbb{S}$ to $\mathbb{R}$. This implies that it can be approximated by a neural network, denoted as α_{NN}. This allows us to represent the value function as $V(b) = \max_{\alpha_{\text{NN}}} b \cdot \alpha_{\text{NN}}$, providing efficient approximation over large or continuous state spaces and direct access to state-wise values $\alpha_{\text{NN}}(s)$ for all $s \in S$. We further define a *Finite Network Controller* (FNC) as follows:

Definition 2 *A Finite Network Controller is a variant of a finite state controller $G = \langle N, n_0, \psi, \eta \rangle$ where: 1. Each node $n \in N$ stores a neural network α_{NN} in addition to the labeled best action; 2. The value of a belief is computed as $V(b) = \max_{n \in G} b \cdot n.\alpha_{NN}$; 3. An FNC can be converted to a standard FSC by removing the α_{NN} in each node.*

The FNC structure facilitates value iteration with sampling-based simulators while enabling compact policy execution: an FNC can be converted into a classical FSC without any loss of execution performance, since the neural networks are only used for value estimation during planning and are not involved in policy execution.

4.2 VALUE ITERATION WITH NEURAL BELLMAN BACKUPS

In this section, we present our main contribution: *Neural Value Iteration* (NVI), which performs value iteration (VI) with a finite network controller for large-scale POMDPs. Such domains are often accessible only through black-box simulators, requiring Monte Carlo sampling for computations. Therefore, we adopt MCVI as the base VI framework and show how our approach integrates into it. The overall procedure for performing value iteration over neural networks is outlined in Algorithm 2, which follows a classical VI scheme: iteratively collecting beliefs and performing backups at those belief points until convergence.

We use a standard belief collection method. Briefly, *CollectBelief* performs a forward tree search according to a reachable belief heuristic, where actions are selected according to the highest upper bound values and observations are selected to maximize the uncertainty between child beliefs' upper and lower bounds. The beliefs in this search trajectory are collected. In our case, the belief update is a particle filtering process, where each belief is represented by a collection of nb_{particle} states. A detailed belief collection algorithm is provided in the appendix, as it is not our main contribution.

Our main algorithmic contribution is presented in Algorithm 3. This *Neural Bellman backup* method defines how to compute a new value function V' for a given belief b in a continuous-state problem:

$$V'(b) = \max_{a \in A} \Bigg\{ \int_{s \in S} R(s,a)\, b(s)\, ds + \gamma \sum_{o \in \Omega} \Pr(o|b,a) \max_{\alpha \in \Gamma} \int_{s' \in S} b^{a,o}(s')\, \alpha(s')\, ds' \Bigg\} \tag{5}$$

Algorithm 3 performs a Bellman backup on a given belief b and updates a finite-network-controller (FNC). It starts with an initialization process (Lines 2-3), and then computes the expected immediate reward (marked in blue in Equation 5) with simulations (between Lines 5-7). Specifically, the algorithm draws particles $s_i \sim b(s)$, where the belief is represented by a particle set, and performs a one-step simulation $(s'_i, o_i, r_i) \sim \text{sim}(s_i, a)$. This produces i.i.d. samples from the joint distribution induced by the belief and the generative model, $p(s', o, r) = \int b(s)\, P(s', o, r \mid s, a)\, ds$. Consequently, the sample average over $\{(s'_i, o_i, r_i)\}$ provides an unbiased Monte Carlo estimate of the corresponding expectations in the Bellman backup, following the same principle as the MC-backup operation used in MCVI.

The key difference between NVI and MCVI lies in how the discounted future return (highlighted in brown in Equation 5) is computed, particularly the evaluation of $\alpha(s')$. Because it's challenging to explicitly represent α-vectors for large or continuous domains, MCVI implicitly represents the value of $\alpha(s')$ by performing Monte Carlo simulations through an FSC policy starting from state s'. This computation becomes very expensive when the problem and FSC size are large. Moreover, to get an accurate approximation of future rewards with stochastic dynamics, MCVI typically requires a large number of simulations. In our case, we explicitly represent each α-vector with a neural network α_{NN} stored at each node, and use it to predict the value of $\alpha(s')$ in Line 9, which significantly reduces computations. Lines 11-16 then compute the discounted expected future

Algorithm 2: Neural Value Iteration

Function `Main` $(b_0, nb_{particle}, nb_{sample}, nb_{sim})$ **:**
  $G \leftarrow \emptyset$
  while $V(b_0)$ *Not Converged* **do**
    $\mathcal{B} \leftarrow$ CollectBeliefs$(b_0, nb_{\text{particle}})$
    foreach $b \in \mathcal{B}$ **do**
      NeuralBackUp$(G, b, nb_{\text{sample}}, nb_{\text{sim}})$
  return G

Algorithm 3: Neural Bellman Backup

1 **Function** `NeuralBackUp` $(G, b, nb_{sample}, nb_{sim})$ **:**
2 For each $a \in A, R_a \leftarrow 0$
3 For each $a \in A, o \in \Omega$, and $n \in G, V_{a,o,n} \leftarrow 0$.
4 **foreach** $a \in A$ **do**
5 **for** $s \sim b$ **do**
6 $s', o, r, \leftarrow \text{Env}_{\text{sim}}(s, a)$
7 $R_a \leftarrow R_a + r$
8 **foreach** $n \in G$ **do**
9 $V_{n,s'} \leftarrow \text{Predict}(n.\alpha_{\text{NN}}, s')$
10 $V_{a,o,n} \leftarrow V_{a,o,n} + V_{n,s'}$
11 **foreach** $o \in \Omega$ **do**
12 $V_{a,o} \leftarrow \max_{n \in G} V_{a,o,n}$
13 $n^*_{a,o} \leftarrow \arg\max_{n \in G} V_{a,o,n}$
14 $V_a \leftarrow \frac{(R_a + \gamma \sum_{o \in \Omega} V_{a,o})}{|b|}$
15 $V^* \leftarrow \max_{a \in A} V_a, \quad a^* \leftarrow \arg\max_{a \in A} V_a$
16 Create a new temporary node n' with action a^* and set edges $(n', n^*_{a^*,o})$ for each $o \in \Omega$
17 **if** *NodeIsUnique*(n', G) **then**
18 $\alpha'_{\text{NN}} \leftarrow \emptyset$
19 $X, Y \leftarrow \text{GenData}(n', G, nb_{\text{sample}}, nb_{\text{sim}})$
20 TrainNN$(\alpha'_{\text{NN}}, X, Y)$
21 Label new node n' with network α'_{NN}
22 $G \leftarrow G \cup \{n'\}$
23 **return** G
24 **Function** `GenData` $(n', G, nb_{sample}, nb_{sim})$ **:**
25 $X, Y \leftarrow \emptyset$
26 **for** $i = 1$ *to* nb_{sample} **do**
27 $s \sim \text{Env}_{sim}$
28 $X \leftarrow X \cup \{s\}$
29 $V_s \leftarrow 0$
30 **if** G *is empty* **then**
31 $V_s \leftarrow \frac{\sum_{i=1}^{\text{nb}_{\text{sim}}} \left(\sum_{t=0}^{\infty} \gamma^t R(s_t^{(i)}, n'.a^*) \right)}{\text{nb}_{\text{sim}}}$
32 **else**
33 $V_s \leftarrow \frac{\sum_{i=1}^{\text{nb}_{\text{sim}}} \left(r^{(i)} + \gamma \cdot V_{n^{(i)}, s'^{(i)}} \right)}{\text{nb}_{\text{sim}}}$
34 $Y \leftarrow Y \cup \{V_s\}$
35 **return** X, Y

return, and we create a new temporary node with the best action found. Compared with MCVI, we additionally include a pruning step before adding the new node to G. This process (function *NodeIsUnique* in Algorithm 3) checks if there exists a node in G with the same best action and node transition edges.

When adding a new node n', a key process is to assign the new node a neural network α_{NN} that explicitly represents an α-vector. We propose to use a separate function GenData to generate training data for α_{NN}. To collect representative states so that α_{NN} can generalize to unseen states, this function first randomly samples nb_{sample} different states from the environment with all possible time depths (Lines 26-27). These nb_{sample} different states are used as the training dataset X. Then for each collected state sample, if the current FNC G is empty, we compute V_s by simulating nb_{sim} rollouts of always performing the same action $n'.a^*$ (Line 31), as there are no transitions between nodes. If G is not empty, we calculate V_s with expected immediate rewards plus the discounted future return $V_{n^{(i)}, s'^{(i)}}$ as shown in Line 33, where $s'^{(i)}, o^{(i)}, r^{(i)} \leftarrow \text{Env}_{\text{sim}}(s, n'.a^*)$ and $n^{(i)} = \eta(G, n', o^{(i)})$. This computation is straightforward because $V_{n^{(i)}, s'^{(i)}}$ simply calls the neural network stored in node $n^{(i)}$ to predict a value such that $V_{n^{(i)}, s'^{(i)}} = n^{(i)}.\alpha_{\text{NN}}(s'^{(i)})$. Finally, all collected V_s values serve as training labels Y. Together, X and Y are used to train α_{NN} and store it in the new node n', and we update the FNC G by adding this new node n'.

In practice, each α_{NN} can be trained efficiently with a small number of sampled states. Simple MLPs are sufficient for the evaluated problems, while reusing sampled states and batching value evaluations further improve efficiency.

Algorithm 3 can be seen as a sampled approximation of Algorithm 1, where exact computations over the full state space are replaced by Monte Carlo simulations. More importantly, it uses neural networks α_{NN} to handle large state spaces instead of the original α-vectors, and the value function is bootstrap-updated with a finite set of neural networks. We also provide a variant of NVI for POMDPs with known models in the appendix, where the transition and observation models are explicitly available rather than being estimated via Monte Carlo simulations.

4.3 DATA COLLECTION STRATEGY

In GenData in Algorithm 3, we sample states from the entire state space rather than from the belief-specific state distribution during each backup. T his choice is motivated by two considerations. First, training only on states sampled from a given belief may cause the neural network to overfit to a limited region of the state space and reduce its generalization ability. Second, broader coverage of the state space enables the α_{NN} representation to generalize to states that may become relevant after policy improvement, which

is important because each α-vector dominates a region of the belief space in the PWLC value function representation. Therefore, sampling from the entire state space provides a more robust training strategy. A possible improvement is a hybrid sampling strategy that combines uniform samples from the state space with samples drawn from the neighborhood of the current belief.

4.4 BENEFITS OF α-VECTOR RECASTING

We analyze the main advantages of recasting α-vectors with neural networks.

A key benefit is the drastic reduction in simulator calls compared to MCVI. Since no explicit α-vector representation is maintained in MCVI, the value $\alpha(s')$ (or $V_{n,s'}$ in Alg. 3) must be estimated via Monte-Carlo rollouts under the current policy starting from node n and state s'. Reliable estimation typically requires many rollouts (nb_{rollout}) with substantial depth (d). Consequently, each MCVI backup loops over actions ($a \in A$), belief particles ($s \in b$), and nodes ($n \in G$), requiring approximately $|A| \times nb_{\text{particle}} \times nb_{\text{node}} \times nb_{\text{rollout}} \times d$ simulator calls. This cost grows with the node set size and becomes prohibitive for large or highly stochastic problems, where nb_{particle}, nb_{rollout}, and d must increase to avoid performance degradation.

In contrast, NVI maintains an explicit neural approximation of α, enabling direct evaluation of $\alpha(s')$ and efficient batched prediction with negligible cost. This reduces simulator usage by orders of magnitude. One may argue that the computation is shifted to data generation and network training. However, only the first NVI backup (when G is empty) requires full rollouts, performed over nb_{sample} states rather than over actions, belief particles, and FSC nodes. Moreover, each NVI backup trains only a small neural network (e.g., an MLP) using $nb_{\text{sample}} \ll |S|$ samples, making the training load very light and easily accelerated on GPUs.

Compared to classic offline solvers such as SARSOP or HSVI, which store α-vectors explicitly, NVI also offers significant memory savings. For example, in RockSample$(20, 20)$ with $|S| > 400$M, storing even a single explicit α-vector requires over 14GB of memory. In contrast, a 3-layer MLP with 1024 neurons per layer requires less than 20MB.

These observations highlight the scalability barriers of prior approaches. Sampling-based methods like MCVI avoid explicit storage but incur heavy Monte-Carlo costs, limiting practicality in large or long-horizon domains [Lim et al., 2011]. Explicit α-vector solvers such as SARSOP and HSVI can achieve ϵ-optimality for small to medium problems, yet their memory requirements grow prohibitively with state space size.

By representing α-vectors as α_{NN}, NVI preserves state-gradient structure with low simulation and memory overhead. We argue that this learning-for-planning paradigm is a key step toward scalable general POMDP planning.

5 EXPERIMENT

We implement our main contribution, NVI, in Julia using the POMDP.jl framework [Egorov et al., 2017]. Planning experiments are conducted on a machine with an Intel i7 3.6GHz CPU, 64GB RAM, and an RTX 4060 GPU, while RL experiments are performed on an NVIDIA A100-SXM4-40GB GPU. In NVI, each FNC node stores an MLP network as α_{NN} to explicitly represent an α-vector. The network takes a vectorized state s as input and outputs the estimated value $V_{n,s}$. Additional experimental details are provided in the appendix.

5.1 BENCHMARK DOMAINS

To evaluate our contribution's performance, we conduct experiments on the following benchmark domains from the POMDP.jl framework. These domains cover the most challenging problems for offline methods, including continuous POMDPs and discrete POMDPs with very large state spaces.

Light Dark The Light Dark problem [Sunberg and Kochenderfer, 2017] involves an agent moving in a 1D world. The agent does not know its initial location and only receives accurate observations in a specific light area. The agent's goal is to localize itself and navigate to a target region.

Lidar Roomba The Lidar Roomba domain [Menda et al., 2023] features a cleaning robot that must localize itself in a 2D environment and reach a goal position. The robot has no knowledge of its initial location and is equipped with a noisy lidar sensor that measures forward distance.

RockSample The RockSample (RS) problem [Smith and Simmons, 2004] is defined by a tuple $\langle n, k \rangle$, where a robot moves in an $n \times n$ grid world containing k rocks. Initially, the robot knows its own location but not the rocks' status (good or bad). The robot can measure rock quality with a noisy sensor whose accuracy depends on the distance to the rock. The goal is to collect as many good rocks as possible before reaching the target location.

Currently, one of the largest POMDP domains tested in recent literature is RS$(20, 20)$, which contains over 419 million states. To our knowledge, no previous offline POMDP solver has successfully solved RS$(20, 20)$, apart from our proposed NVI approach.

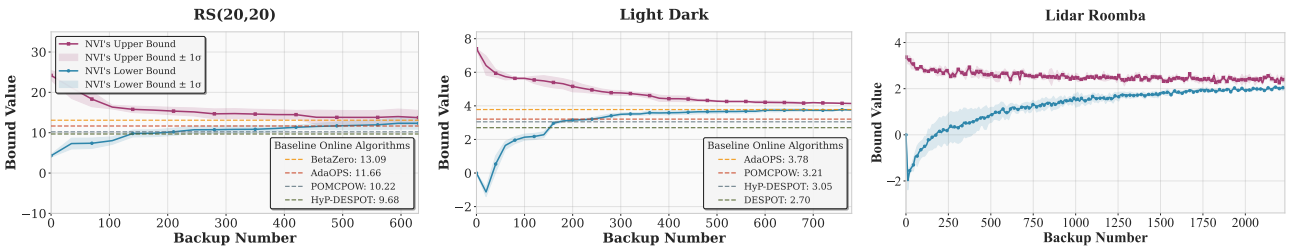


Figure 2: Upper and lower bound estimates of NVI over backup iterations in RS$(20, 20)$, Light Dark, and Lidar Roomba. The lower bounds are obtained from the value iteration process in Algorithm 2, while the upper-bound update procedure is described in the Appendix. Baseline online planning methods [Somani et al., 2013, Sunberg and Kochenderfer, 2017, Cai et al., 2021, Wu et al., 2021] are reported from BetaZero [Moss et al., 2024].

5.2 OFFLINE POMDP METHODS

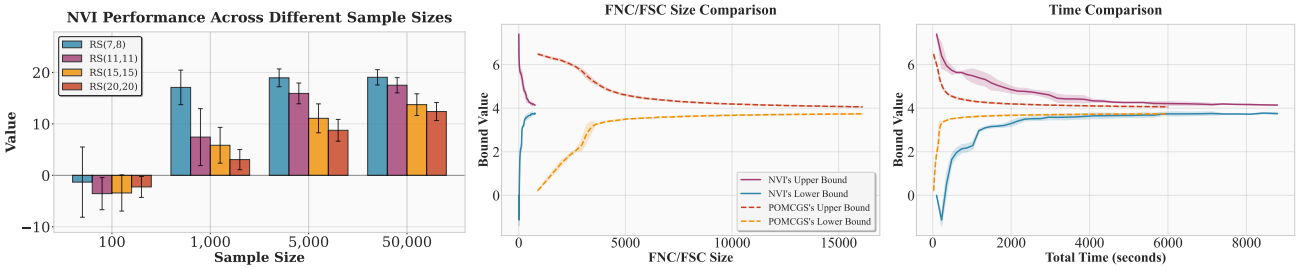
We compare NVI with several state-of-the-art offline POMDP methods including SARSOP, MCVI, and POMCGS, along with a reinforcement learning baseline (PPO with recurrent neural network). For all planning methods, we set time limits of: • 1 hour for small problems (RS$(7, 8)$); • 6 hours for medium problems (RS$(11, 11)$ and Light Dark); • 24 hours for large problems (RS$(15, 15)$ and RS$(20, 20)$ and Lidar Roomba). No time limit is applied to PPO. Among all tested algorithms, only SARSOP requires explicit models providing exact state transition and observation probabilities. Note that model loading and algorithm initialization times (which can be substantial, e.g., many hours for SARSOP) are excluded from these limits. For continuous domains, all sampling-based offline solvers require discretization; the details are provided in Table 2.

Parameter Analysis NVI's planning process involves three key parameters: nb_{sample}, which controls how many sampled states are used to approximate the full state space during the training of α_{NN}; nb_{particle}, which determines how many particles are used to represent a belief; and nb_{sim}, which sets the number of Monte Carlo simulations performed per particle to estimate expected outcomes. The parameters nb_{particle} and nb_{sim} are common to most Monte Carlo-based planners, both online and offline, as these methods rely on particle filtering and simulation for belief tracking and value estimation. Thus, we omit a detailed discussion of these parameters and focus our analysis on nb_{sample}, which is only used in NVI among current offline approaches.

As shown in Figure 3a, increasing nb_{sample} generally improves NVI's performance. For smaller domains such as RS$(7, 8)$, a relatively small nb_{sample} suffices to train α_{NN} effectively and enable generalization to unseen states. If nb_{sample} exceeds the size of the state space $|S|$, α_{NN} essentially memorizes the entire state space, behaving more like a tabular representation. For larger domains, a higher nb_{sample} is typically needed to achieve good generalization. However, even then, $nb_{\text{sample}} \ll |S|$, underscoring the efficiency of using neural networks to approximate α-vectors.

Performance Comparison Performance results are reported in Table 1. On small to moderate-sized problems, SARSOP achieves the highest returns owing to its exact model-based Bellman backups. However, despite obtaining high returns on RS$(11, 11)$, SARSOP remains impractical due to its substantial memory requirements and long initialization time. For example, solving RS$(11, 11)$ requires approximately 14,GB of RAM, and loading and initializing the domain takes more than 24 hours. These limitations, together with the computational cost of performing exact Bellman backups over the entire state space $\mathbb{S}$, prevent SARSOP from scaling to larger problems. On the other hand, sampling-based approaches demonstrate better scalability for large and continuous domains. Notably, among these methods, only NVI successfully solves the extremely large RS(20,20) domain with over 419 million states. Although MCVI should theoretically achieve performance similar to that of SARSOP or NVI given sufficient Monte Carlo simulations and particle sizes, its lack of explicit α vector representation dramatically increases the computational requirements for accurate estimation in large problems. Another interesting observation is that despite using simple discretization schemes for continuous domains (Light Dark and Lidar Roomba), sampling-based methods such as POMCGS and NVI perform relatively well on these problems. This suggests that continuous domains may not inherently pose significant challenges for offline planning, as simple discretization can effectively convert them to small discrete domains. However, even discretized domains like RS$(20, 20)$ with only 3 observations remain extremely challenging due to their enormous state spaces.

Additionally, since these benchmarks are well-studied in the online planning community, we compare NVI's upper and lower bounds during backups with state-of-the-art online methods reported in the literature. As shown in Figure 2, NVI's bounds gradually converge toward near-optimal solutions, consistent with the performance of online planners. We omit the comparison for the Lidar Roomba problem,



(a) Impact of nb_{sample} in RS domains (b) NVI vs POMCGS: FNC/FSC size (c) NVI vs POMCGS: Planning time

Figure 3: NVI analysis: (a) Effect of nb_{sample} on RS; (b, c) Comparison with POMCGS in FSC size and planning time on Light Dark domain, NVI's FNC is converted to a FSC

	RS$(7,8)$	RS$(11,11)$	RS$(15,15)$	RS$(20,20)$	Light Dark	Lidar Roomba
$\lvert\mathcal{S}\rvert$	$12,544$	$247,808$	$7,372,800$	$419,430,400$	∞	∞
$\lvert\mathcal{A}\rvert$	13	16	20	25	3	∞
$\lvert\mathcal{Z}\rvert$	3	3	3	3	∞	∞
Recurrent PPO	7.73 ± 0.01	6.30 ± 0.04	6.16 ± 2.88	3.97 ± 0.02	3.26 ± 0.01	-2.07 ± 0.02
SARSOP*	$\mathbf{21.57 \pm 1.82}$	$\mathbf{19.09 \pm 2.40}$	—	—	—	—
POMCGS	20.17 ± 1.68	18.53 ± 1.04	13.45 ± 1.66	3.91 ± 0.83 (†)	$\mathbf{3.74 \pm 0.06}$	1.08 ± 0.19
MCVI	17.34 ± 2.52	13.78 ± 2.13	10.78 ± 1.92	4.57 ± 2.19	2.94 ± 0.15	0.68 ± 0.16
NVI	18.95 ± 1.49	17.51 ± 1.56	$\mathbf{13.60 \pm 2.27}$	$\mathbf{12.31 \pm 1.65}$	3.73 ± 0.10	$\mathbf{2.03 \pm 0.03}$

Table 1: Performance comparison of different offline POMDP algorithms. The symbol "†" indicates out-of-memory errors during computation, and "—" denotes that the algorithm is not applicable to the corresponding domain. All methods are evaluated over five independent runs, with policies from sampling-based approaches evaluated using 10^5 simulations per run. In RockSample domains, each run uses a different problem instance with randomized rock positions and statuses, while the robot always starts at position $(1,1)$ to maximize the distance to the exit area. The asterisk "*" for SARSOP indicates that it is the only method requiring an explicit POMDP model for exact computation; all other methods operate with black-box simulators.

as we didn't find competitive performance in the current literature.

Last but not least, despite an extensive parameter search for Recurrent PPO (as detailed in the appendix), it failed to produce competitive policies compared to offline planning methods in most domains. Notably, while some individual episodes during training showed promising behavior, the learning process frequently converged to a local optimum corresponding to a myopic solution (e.g., in RockSample, the agent moves directly to the exit area). Notably, A higher performance in the RockSample problem is reported in [Tao et al., 2025], which may be attributed to differences in parameters and problem configurations.

5.3 A CLOSER LOOK AT SAMPLING-BASED APPROACHES

We compare NVI and POMCGS on the Light Dark domain, where both achieve similar performance. As shown in Figures 3b and 3c, POMCGS finds reasonable solutions much faster due to its top-down search and belief node merging. However, because it does not follow a value iteration scheme, its FSC nodes do not correspond to α-vectors that generalize over broad belief regions, which leads to large and memory-intensive controllers in complex domains. In contrast, NVI performs value iterations and exploits the PWLC structure, resulting in more compact policies. Furthermore, NVI doesn't require state discretization, as it does not rely on belief comparison.

6 DISCUSSION

Contribution In this article, we demonstrate that a POMDP value function can be represented by a finite set of neural networks. Based on this insight, we introduce the *finite network controller* (FNC), a variant of the finite state controller in which each node stores a neural network to explicitly represent an α-vector. We further propose *Neural Value Iteration* (NVI), a new offline planning algorithm that follows the classical value iteration framework while performing Bellman backups directly on neural networks. Empirically, we show that NVI can tackle highly challeng-

Problem (Type)	MCVI & NVI	POMCGS
Light Dark (C,D,C)		
States	—	State Grid $= [1.0]$
Actions	—	—
Observations	KMeans (20)	KMeans (20)
Lidar Roomba (C,C,C)		
States	—	State Grid (4002)
Actions	Subset (9)	Progressive Widening
Observations	KMeans (10)	KMeans (10)

Table 2: Discretization approaches used by sampling-based methods. Notation: C = Continuous, D = Discrete (state/action/observation spaces). Numbers in parentheses indicate the corresponding discrete sizes, and "—" denotes that no discretization is required. "Subset()" uniformly samples a subset of actions from the continuous action space, while "KMeans" discretizes the continuous observation space using clustering. The state grid is only required by POMCGS to compute L1 distances between beliefs using discretized bins; NVI and MCVI do not require state discretization.

ing problems such as RS$(20, 20)$, with over 419 million states—well beyond the limits of existing offline solvers.

Convergence, Accuracy and α_{NN} Generalization Since NVI follows the classical point-based value iteration framework, similar convergence properties [Smith and Simmons, 2005, Bai et al., 2010] can be expected in the limit under the assumption of no approximation error. A natural concern is whether imperfect training of α_{NN} may result in poor generalization to unseen states. In practice, insufficient fitting may be reflected by a high training loss, while poor generalization manifests as inaccurate value predictions on unseen states, which can result in suboptimal action selection and degraded performance. These effects can be assessed through execution analysis or comparison with state-of-the-art planners, as shown in Figure 2. Empirically, we do not observe such generalization issues in NVI with a sufficient number of training states. When suboptimal performance occurs, it is mainly attributed to belief approximation errors from particle-based belief representations and estimation errors introduced by Monte Carlo simulations during Bellman backups. We view establishing formal error bounds and developing principled strategies for selecting the nb_{sample} training states as important directions for future work.

Limitation The main limitation of NVI is that the current approach still requires discretization of actions and observations when applied to POMDPs with continuous action or observation spaces.

Future Work The idea of representing α-vectors with neural networks opens several directions for future research. Theoretically, error bounds for the neural representation can be formalized and analyzed. Practically, techniques such as improved state sampling [Brechtel et al., 2013], belief selection heuristics, macro actions [Lim et al., 2011], and observation grouping can be integrated into Neural Value Iteration to further enhance the performance in complex or continuous domains.

7 CONCLUSION

Offline point-based value iteration methods progressed from classic algorithms such as PBVI to HSVI and SARSOP, and later to MCVI; however, their scalability in very large domains has remained fundamentally limited by memory and computational costs. Over the past decade, compared to the rapid advancement of online planning methods and deep learning approaches, these classic offline POMDP approaches have gradually received less attention due to scalability issues. In this article, we show that advancements in neural networks can also bring new life to the classic point-based value iteration scheme. Specifically, by reinterpreting α-vectors as neural networks and leveraging state-gradient information to scale efficiently to large state spaces, this new perspective may open a new direction for deep offline POMDP planning, highlighting that structured, model-based decision making still holds powerful potential. In addition, many existing techniques from traditional value iteration methods can be naturally extended to Neural Value Iteration.

Acknowledgements

This work has been supported by the EPSRC Energy Programme under UKAEA/EPSRC Fusion Grant 2022/2027 No. EP/W006839/1.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *SIGKDD*, pages 2623–2631, 2019.

Haoyu Bai, David Hsu, Wee Sun Lee, and Vien A Ngo. Monte carlo value iteration for continuous-state pomdps. In *Algorithmic Foundations of Robotics IX: Selected Contributions of the Ninth International Workshop on the Algorithmic Foundations of Robotics*, pages 175–191. Springer, 2010.

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.

Sebastian Brechtel, Tobias Gindele, and Rüdiger Dillmann. Solving continuous pomdps: Value iteration with incremental learning of an efficient space representation. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 370–378, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

Panpan Cai, Yuanfu Luo, David Hsu, and Wee Sun Lee. Hyp-despot: A hybrid parallel algorithm for online planning under uncertainty. *The International Journal of Robotics Research*, 40(2-3):558–573, 2021.

Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. Pomdps.jl: A framework for sequential decision making under uncertainty. *JMLR*, 18(26):1–5, 2017.

Eric Hansen. An improved policy iteration algorithm for partially observable MDPs. *Advances in neural information processing systems (NIPS)*, 10, 1997.

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

Peter Karkus, David Hsu, and Wee Sun Lee. Qmdp-net: Deep learning for planning under partial observability. *Advances in neural information processing systems*, 30, 2017.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML-06*, 2006.

Hanna Kurniawati. Partially Observable Markov Decision Processes and Robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(1):253–277, 2022.

Hanna Kurniawati, David Hsu, and Wee Sun Lee. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *RSS-08*, 2008.

Mikko Lauri, David Hsu, and Joni Pajarinen. Partially observable markov decision processes in robotics: A survey. *IEEE Transactions on Robotics*, 39(1):21–40, 2022.

Zhan Wei Lim, David Hsu, and Wee Sun Lee. Monte carlo value iteration with macro-actions. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'11, page 1287–1295, Red Hook, NY, USA, 2011. Curran Associates Inc.

Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *AAAI-99*, 1999.

K. Menda, Z. Sunberg, and M. Kochenderfer. RoombaPOMDPs.jl. `https://github.com/sisl/RoombaPOMDPs.jl/`, 2023. [Online; accessed 03-May-2025].

Robert J. Moss, Anthony Corso, Jef Caers, and Mykel Kochenderfer. Betazero: Belief-state planning for long-horizon POMDPs using learned approximations. In *Reinforcement Learning Conference*, 2024.

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

Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI-03*, 2003.

Josep M Porta, Nikos Vlassis, Matthijs TJ Spaan, and Pascal Poupart. Point-based value iteration for continuous pomdps. *Journal of Machine Learning Research*, 7(Nov): 2329–2367, 2006.

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

Guy Shani, Pascal Poupart, Ronen I Brafman, and Solomon Eyal Shimony. Efficient add operations for point-based algorithms. In *ICAPS-08*, pages 330–337, 2008.

Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems (AAMAS)*, 27:1–51, 2013.

David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *NIPS-10*, volume 23, 2010.

Hyeong Seop Sim, Kee-Eung Kim, Jin Hyung Kim, Du-Seong Chang, and Myoung-Wan Koo. Symbolic heuristic search value iteration for factored pomdps. In *AAAI*, pages 1088–1093, 2008.

Trey Smith and Reid Simmons. Heuristic search value iteration for POMDPs. In *UAI-04*, 2004.

Trey Smith and Reid Simmons. Point-based POMDP algorithms: Improved analysis and implementation. In *UAI-05*, 07 2005.

Adhiraj Somani, Nan Ye, David Hsu, and Wee Sun Lee. DESPOT: Online POMDP planning with regularization. In *NIPS-13*, 2013.

Edward Jay Sondik. *The optimal control of partially observable Markov processes*. Stanford University, 1971.

Zachary Sunberg and Mykel Kochenderfer. POMCPOW: An online algorithm for POMDPs with continuous state, action, and observation spaces. In *ICAPS-17*, 09 2017.

Ruo Yu Tao, Kaicheng Guo, Cameron Allen, and George Konidaris. Benchmarking partial observability in reinforcement learning with a suite of memory-improvable domains. *The Reinforcement Learning Journal*, 2025. URL `http://github.com/taodav/pobax`.

Chenyang Wu, Guoyu Yang, Zongzhang Zhang, Yang Yu, Dong Li, Wulong Liu, and Jianye Hao. Adaptive online packing-guided search for POMDPs. In *NIPS-21*, 2021.

Yang You, Vincent Thomas, Alex Schutz, Robert Skilton, Nick Hawes, and Olivier Buffet. Partially observable monte-carlo graph search. In *ICAPS-25*, volume 35, pages 279–287, 2025.

Neural Value Iteration - Supplementary Material

Yang You[1] **Ufuk Çakır**[2] **Alex Schutz**[2] **Nick Hawes**[2]

[1]United Kingdom Atomic Energy Authority,
[2]Oxford Robotics Institute, University of Oxford,

A NEURAL VALUE ITERATION WITH KNOWN POMDP MODELS

Neural Value Iteration (NVI) can also be applied to POMDPs with known transition and observation models. In this case, the core backup is shown in Algorithm 4, for each action, we first construct a supervised dataset of state-value pairs $\{(s_i, \alpha_a(s_i))\}_{i=1}^N$, where $\alpha_a(s_i)$ is computed directly from the known transition and observation models, and predicted future value $\alpha_{a,o,\mathrm{NN}}(s')$ according to the Bellman backup. A neural network $\alpha_{a,\mathrm{NN}}$ is then trained to approximate α_a. Finally, the action whose corresponding network maximizes the value of the current belief is selected, and the resulting network is added to Γ_{NN}.

Unlike the simulator-based version, Monte Carlo simulation is no longer required to approximate the Bellman backup, since the expectations can be evaluated directly from the known models. However, sampled states are still recommended to construct a supervised training dataset for fitting the neural network representation of the α-function, especially in large state spaces.

B BELIEF COLLECTION PROCESS

In our work, we adopt a belief collection strategy similar to HSVI [Smith and Simmons, 2005] during value iteration. Specifically, at the current belief b, the algorithm selects the action

$$a = \arg\max_a Q_{\overline{V}}(b, a),$$

where $Q_{\overline{V}}(b, a)$ is the action value induced by the upper bound. The next observation is selected according to the maximum excess uncertainty:

$$o = \arg\max_o \Pr(o|b, a) \left(\overline{V}(b^{a,o}) - \underline{V}(b^{a,o})\right).$$

Beliefs are collected along this trajectory until the desired search depth is reached. The upper bound is initialized using the standard MDP relaxation. During the backup process, the upper bound is updated by

$$\overline{V} \leftarrow \overline{V} \cup \{(b, H\overline{V}(b))\},$$

where

$$H\overline{V}(b) = \max_a \left[R(b, a) + \gamma \sum_o \Pr(o|b, a)\overline{V}(\tau(b, a, o)) \right].$$

With exact computation, this procedure preserves the upper-bound property:

$$\overline{V}(b) \geq V^*(b)$$

for all reachable beliefs, as shown in HSVI and SARSOP.

Algorithm 4: Neural Bellman Backup for POMDPs with Known Models

Function `Backup` (Γ_{NN}, b, N) **:**
  foreach $a \in A$ *and* $o \in \Omega$ **do**
    $\alpha_{a,o,\text{NN}} \leftarrow \arg\max_{\alpha_{\text{NN}} \in \Gamma_{\text{NN}}} (\alpha_{\text{NN}} \cdot \tau(b, a, o))$
  foreach $a \in A$ **do**
    Sample N states $\{s_i\}_{i=1}^{N}$ from a state sampling distribution;
    Construct the dataset $\{(s_i, \alpha_a(s_i))\}_{i=1}^{N}$, where

$$\alpha_a(s_i) = R(s_i, a) + \gamma \sum_{o \in \Omega} \sum_{s' \in S} T(s_i, a, s') O(s', a, o) \alpha_{a,o,\text{NN}}(s').$$

    Train $\alpha_{a,\text{NN}}$ using the constructed dataset;
  $\alpha'_{\text{NN}} \leftarrow \arg\max_{a \in A} (\alpha_{a,\text{NN}} \cdot b)$;
  return $\Gamma_{\text{NN}} \cup \{\alpha'_{\text{NN}}\}$.

In our sampling-based implementation, the upper bound is updated at the collected belief nodes in $\mathcal{B}_0$, which are obtained through a forward search process starting from the initial belief. Our belief collection procedure is provided in Algorithm 5. However, our sampling-based approach approximates belief updates and value backups using particle filtering and Monte Carlo simulations, which introduces estimation errors. Since the estimated value bounds guide both action and observation selection, these errors may lead to suboptimal exploration trajectories and degrade the quality of the resulting policy.

C RL EXPERIMENTS

For the RL experiments, we use the pobax package [Tao et al., 2025], which is entirely written in JAX [Bradbury et al., 2018][1]. RL experiments require significant amounts of fine-tuning hyperparameters, which is why JAX is currently the state-of-the-art choice, since it can run multiple runs in parallel on the GPU.

All experiments were performed on a Rocky Linux 9.6 (Blue Onyx) system with one NVIDIA A100-SXM4-40GB GPU and CUDA 12.9.

The network architecture of the actor critic model is the same as in [Tao et al., 2025], which is shown in Fig. 4. We performed hyperparameter tuning for each environment using the optuna package [Akiba et al., 2019] with 50 trials and the following ranges:

- Learning rate: lr $\in [10^{-5}, 10^{-3}]$ (log-uniform),
- Clipping parameter: $\epsilon \in [0.1, 0.4]$,
- Entropy coefficient: entropy_coeff $\in [0.0, 0.1]$,
- Value function coefficient: vf_coeff $\in [0.3, 0.7]$,
- GAE parameter: $\lambda_0 \in [0.1, 0.99]$,
- Gradient clipping: max_grad_norm $\in [0.5, 2.0]$,
- Parallel environments: num_envs $\in \{64, 128, 256\}$,
- Rollout length: num_steps $\in \{64, 128, 256\}$,
- Hidden layer size: hidden_size $\in \{128, 256, 512, 1024\}$,
- PPO update epochs: update_epochs $\in \{2, 3, \ldots, 8\}$,
- Minibatches: num_minibatches $\in \{2, 4, 8, 16\}$.

The best hyperparameters were chosen from the optuna results and are shown in table 3.

[1] Commit 4e53aac is the latest version of the code we used

Algorithm 5: CollectBeliefs

Input : $\mathcal{T}$: the belief tree
$n_{\mathcal{T}}$: current belief tree node
ϵ: uncertainty threshold
γ: discount factor
d: current depth
L: maximum depth
nb_{particle}: number of particles
V_{mdp}: MDP heuristic
$\mathcal{B}$: output list of sampled beliefs

Output : $\mathcal{B}$: collected beliefs

if $\overline{V}(b_0) - \underline{V}(b_0) > \epsilon$ ***and*** $d < L$ **then**
  $a \leftarrow \arg\max_a Q_{\overline{V}}(n_{\mathcal{T}}, a)$;
  foreach $o \in \Omega$ **do**
    $b' \leftarrow \text{ParticleFilter}(nb_{\text{particle}}, n_{\mathcal{T}}.b, a, o)$;
    if $b' \notin \mathcal{T}$ **then**
      Add b' to $\mathcal{T}$ as a child of $n_{\mathcal{T}}$ via edge (a, o) ;
      Initialize $\overline{V}(b')$ using the QMDP heuristic with V_{mdp} ;
  $o \leftarrow \arg\max_{o \in \Omega}(\Pr(o|n_{\mathcal{T}}.b, a) \cdot \left(\overline{V}(b^{a,o}) - \underline{V}(b^{a,o}) - \epsilon\gamma^{-(d+1)}\right))$;
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{b^{a,o}\}$;
  Recursively call CollectBeliefs on the child node $n'_{\mathcal{T}}$ reached via edge (a, o) from $n_{\mathcal{T}}$ with updated depth $d \leftarrow d + 1$;
return $\mathcal{B}$

For the RockSample environments, we used the implemented pobax implementations. Since there exists no JAX implementation of lightdark and roomba yet, we ported those from the Julia equivalents. The full discounted reward during training is shown in Figure 2.

D PLANNING EXPERIMENTS

There are three planning algorithms compared in our experiments: SARSOP, POMCGS, and MCVI. For POMCGS, we used the official POMCGS Julia implementation available at `https://github.com/ori-goals/POMCGraphSearch.jl`. For MCVI, we implemented a customized version in Julia using the POMDP.jl framework. We applied several custom optimizations, including vectorized operations, multi-threading, and static value types to accelerate computation. These improvements enabled faster performance compared to existing Julia implementation (e.g., `https://github.com/JuliaPOMDP/MCVI.jl`) in our tested domains.

For SARSOP, we used the original C++ implementation available at `https://github.com/AdaCompNUS/sarsop`. POMDP model files were exported in the `POMDPX` format for compatibility with SARSOP.

We used default settings for SARSOP and omit further parameter details. For the sampling-based methods (POMCGS, MCVI, and NVI), all share two key parameters: n_{particle}, the number of particles used to represent beliefs, and n_{sim}, the number of simulations used for each state particle to approximate expectations. We tested the following ranges:

- $nb_{\text{particle}} \in \{1000, 5000, 10000, 20000, 50000\}$
- $nb_{\text{sim}} \in \{10, 50, 100, 500\}$

We found that $nb_{\text{particle}} = 10000$ and $nb_{\text{sim}} = 100$ were sufficient for good belief representation and expectation approximation across most benchmarks. These values were used in our final experiments.

For NVI, an additional parameter nb_{sample} is used to sample the state space, and its impact is discussed in the main paper.

Network Architecture Selection In NVI, the POMDP value function is approximated by a finite set of neural networks, each denoted as α_{NN}. In our experiments, we trained all α_{NN} with multilayer perceptrons (MLPs) with ReLU activations.

```
ActorCritic(
  Actor(
    Sequential(
        (0): Dense(in_dims=input_dim, out_dims=hidden_size, bias=True)
        (1): ReLU()
        (3): GRU(in_dims=hidden_size, hidden_size=hidden_size)
        (4): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
        (5): ReLU()
        (6): Dense(in_dims=hidden_size, out_dims=action_dims, bias=True)
        (7): Categorical() or MultivariateNormalDiag()
    )
  )
  Critic(
    Sequential(
        (0): Dense(in_dims=input_dim, out_dims=hidden_size, bias=True)
        (1): ReLU()
        (3): GRU(in_dims=hidden_size, hidden_size=hidden_size)
        (4): Dense(in_dims=hidden_size, out_dims=hidden_size, bias=True)
        (5): ReLU()
        (6): Dense(in_dims=hidden_size, out_dims=1, bias=True)
    )
  )
)
```

Figure 4: Actor critic architecture similar as in [Tao et al., 2025]

We evaluated several candidate architectures with varying depth and width to balance approximation capacity and computational efficiency. The tested configurations included:

```
Dense(input_dim, 64, relu)
Dense(64, 64, relu)
Dense(64, 1)

Dense(input_dim, 128, relu)
Dense(128, 64, relu)
Dense(64, 32, relu)
Dense(32, 1)

Dense(input_dim, 128, relu)
Dense(128, 64, relu)
Dense(64, 64, relu)
Dense(64, 1)

Dense(input_dim, 256, relu)
Dense(256, 128, relu)
Dense(128, 64, relu)
Dense(64, 1)

Dense(input_dim, 512, relu)
Dense(512, 256, relu)
Dense(256, 128, relu)
Dense(128, 64, relu)
Dense(64, 1)
```

Table 3: Best hyperparameters for PPO on various environments.

Parameter	RS$(7,8)$	RS$(11,11)$	RS$(15,15)$	RS$(20,20)$	Light Dark	Lidar Roomba
Final Return	7.74	6.30	6.16	3.97	3.26	-2.07
Learning Rate (α)	1.41e-4	5.08e-4	1.54e-4	2.42e-4	8.99e-4	3.81e-4
Clip Epsilon (ϵ)	0.284	0.389	0.319	0.227	0.1024	0.3107
Entropy Coeff. (β)	0.084	0.015	0.076	0.0004	0.0126	0.0566
Value Function Coeff. (c_{vf})	0.572	0.318	0.599	0.435	0.4793	0.4785
GAE λ_0	0.367	0.448	0.102	0.734	0.1	0.1277
Max Grad Norm	1.064	0.786	1.186	0.558	0.5627	1.9196
Num Envs	64	64	128	128	64	64
Num Steps	64	64	64	64	64	256
Hidden Size	256	128	256	128	512	256
Update Epochs	3	6	8	6	7	6
Num Minibatches	8	8	4	16	16	8

Empirically, increasing network size beyond a moderate capacity did not produce noticeable improvements in policy quality. In contrast, smaller MLPs achieved comparable performance while reducing computational cost. In practice, the training time per backup was typically below 5 seconds, including CPU–GPU data transfer overhead with small MLPs.

Based on this empirical trade-off, we selected the following architectures:

- **RockSample:**

```
Dense(input_dim, 128, relu)
Dense(128, 64, relu)
Dense(64, 64, relu)
Dense(64, 1)
```

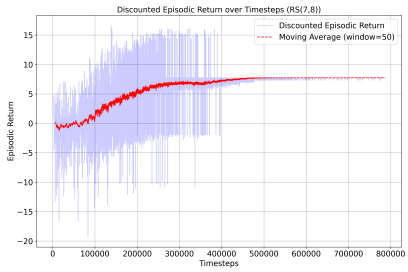
- **Light-Dark and Lidar Roomba:**

```
Dense(input_dim, 128, relu)
Dense(128, 64, relu)
Dense(64, 32, relu)
Dense(32, 1)
```

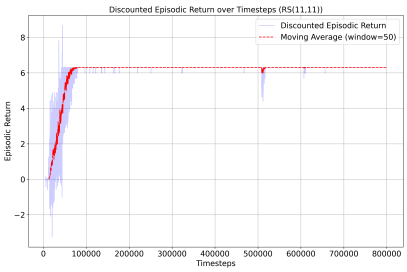
We fixed the learning rate to 0.005 and used a batch size of 1024 for training each α_{NN}. Training stops early if the mean squared error (MSE) falls below 0.1 or after a maximum of 10000 epochs.

Input and Output of Network For each α_{NN}, the input is a vectorized state representation and the output is a scalar, consistent with the definition of an α-vector as a mapping from $\mathbb{S}$ to $\mathbb{R}$. Specifically, in RockSample, the input consists of a one-hot encoding of the robot position concatenated with the rock status vector. In LightDark and LidarRoomba, the input is the state vector representing the agent's location.

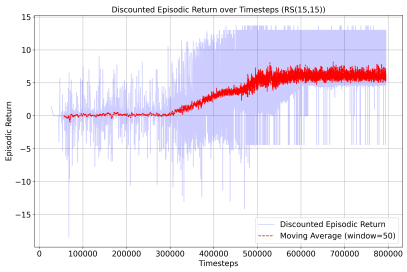
Reproducibility We provide our source code in the Supplementary Material, along with Jupyter notebooks that guide users to reproduce and understand each experiment step-by-step.



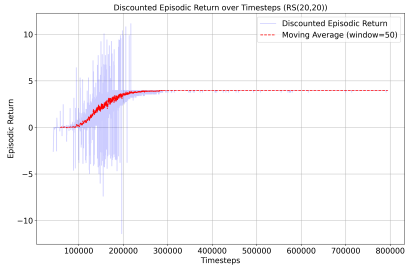
(a) RS(7,8)



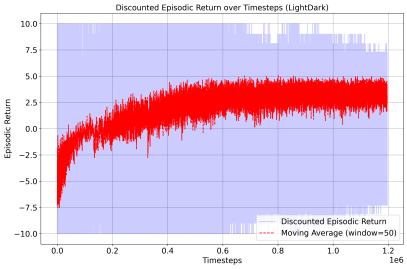
(b) RS(11,11)



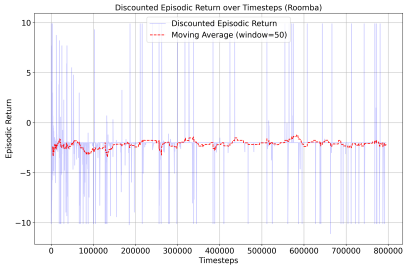
(c) RS(15,15)



(d) RS(20,20)



(e) Light-Dark



(f) Lidar-Roomba

Figure 5: Training-progress curves for all evaluated environments.