POSGGym: A Library for Decision-Theoretic Planning and Learning in Partially Observable, Multi-Agent Environments

Jonathon Schwartz[1], **Rhys Newbury**[1,2], **Dana Kulić**[2], **Hanna Kurniawati**[1],

[1]School of Computing, Australian National University, Canberra, ACT, Australia
[2]Department of Electrical and Computer Systems Engineering, Monash University, Clayton, VIC, Australia
jonathon.schwartz@anu.edu.au, rhys.newbury@anu.edu.au, dana.kulic@monash.edu, hanna.kurniawati@anu.edu.au

Abstract

Seamless integration of Planning Under Uncertainty and Reinforcement Learning (RL) promises to bring the best of both model-driven and data-driven worlds to multi-agent decision-making, resulting in an approach with assurances on performance that scales well to more complex problems. Despite this potential, progress in developing such methods has been hindered by the lack of adequate evaluation and simulation platforms. Researchers have had to rely on creating custom environments, which reduces efficiency and makes comparing new methods difficult. In this paper, we introduce POSGGym: a library for facilitating planning and RL research in partially observable, multi-agent domains. It provides a diverse collection of discrete and continuous environments, complete with their dynamics models and a reference set of policies that can be used to evaluate generalization to novel partners. Leveraging POSGGym, we empirically investigate existing state-of-the-art planning methods and a method that combines planning and RL in the type-based reasoning setting. Our experiments corroborate that combining planning and RL can yield superior performance compared to planning or RL alone, given the model of the environment and other agents is correct. However, our particular setup also reveals that this integrated approach could result in worse performance when the model of other agents is incorrect. Our findings indicate the benefit of integrating planning and RL in partially observable, multi-agent domains, while serving to highlight several important directions for future research. Code available at: https://github.com/RDLLab/posggym.

1 Introduction

Decision-theoretic planning, also known as *planning under uncertainty*, addresses the problem of using a dynamics model to find the optimal way to behave in uncertain environments (Blythe 1999; Boutilier, Dean, and Hanks 1999). In scenarios involving a single-agent, this is formalized by the partially observable Markov decision process (POMDP) (Kaelbling, Littman, and Cassandra 1998), which incorporates uncertainty in the state of the environment and the stochastic outcomes of actions. Extending to the multi-agent setting, various approaches exist (Seuken and Zilberstein 2008), with the Partially Observable Stochastic Game (POSG) (Hansen, Bernstein, and Zilberstein 2004) being one of the most general. The appeal of decision-theoretic planning lies in its mathematical rigor and theoretical guarantees for optimal decision-making under uncertainty, vital for reliable autonomous systems in domains like robotics (Kurniawati 2022; Woodward and Wood 2012), autonomous driving (Carr et al. 2021), and security (Seymour and Peterson 2009; Ng et al. 2010).

Unfortunately, applying planning under uncertainty to large, multi-agent environments has remained a challenge, due to difficulties around computational complexity and handling of complex dynamics models. Advances in deep learning (LeCun, Bengio, and Hinton 2015) offer promising solutions to alleviate these difficulties. Deep learning has been able to leverage growing compute much more effectively than planning to solve increasingly difficult problems (Kaplan et al. 2020). Combining deep reinforcement learning (RL) with planning has already led to remarkable achievements in multi-agent decision making in games (Silver et al. 2018; Brown et al. 2020; Lerer et al. 2020). Integrating learning with planning presents an exciting opportunity for the creation of robust autonomous agents capable of scaling to complex environments while effectively handling uncertainty.

In this work, we introduce *POSGGym*, a research library for planning and RL in partially observable, multi-agent environments (Figure 1). POSGGym models environments using the POSG framework and supports both discrete and continuous domains. It includes implementations of established and newer planning benchmarks, all under a unified API compatible with the existing ecosystem of multi-agent RL (MARL) libraries. Additionally, POSGGym provides a reference set of co-players, crucial for multi-agent research and for enabling reproducible evaluation with a range of partners.

POSGGym enables extensive empirical investigation into decision-theoretic planning and its combination with RL. We evaluate four state-of-the-art planners and integrated planning and RL across a diverse set of discrete environments. Our evaluation encompasses performance with both known and unknown co-player populations, allowing us to study various properties of each method along with generalization to novel partners. Our results demonstrate how decision-theoretic planning and RL can be combined effectively to get better performance than either method alone.

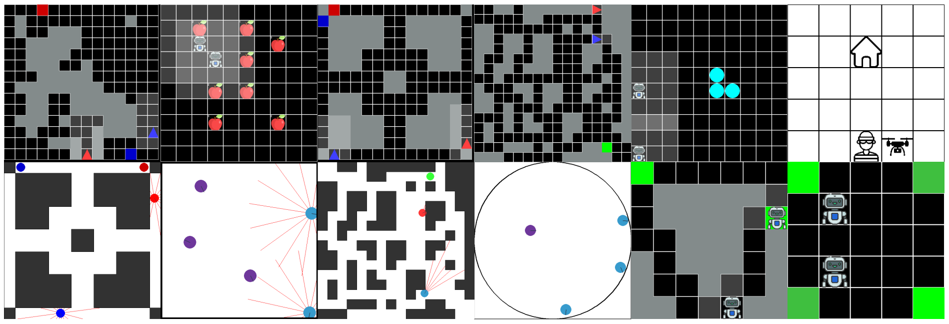


Figure 1: POSGGym is a library for planning and learning research in partially observable, multi-agent domains. It includes a diverse set of discrete and continuous environments along with a collection of reference policies for reproducible evaluation.

Furthermore, our findings show areas where existing planning methods perform poorly and highlight promising directions for future research.

2 Background

2.1 Partially Observable Stochastic Games

Partially Observable Stochastic Games (POSGs) (Hansen, Bernstein, and Zilberstein 2004) generalize various other formal decision-making models. A Partially Observable Markov Decision Process (POMDP) is a POSG with a single agent (Kaelbling, Littman, and Cassandra 1998). A decentralized-POMDP (Dec-POMDP) (Bernstein et al. 2002) is a fully cooperative POSG where all agents share a reward function. A Markov game, also known as a stochastic game, is a fully observable POSG (Shapley 1953). The generality of POSGs has lead to their wide use in MARL and decision-theoretic planning.

Formally, a POSG is a tuple $\mathcal{M} = \langle \mathcal{I}, \mathcal{S}, S_0, \vec{\mathcal{A}}, \vec{\mathcal{O}}, \mathcal{T}, \mathcal{Z}, \mathcal{R} \rangle$ consisting of N agents indexed $\mathcal{I} = \{1, \dots, N\}$, a set of environment states $\mathcal{S}$, an initial state distribution $S_0 \in \Delta(\mathcal{S})$, the joint action space $\vec{\mathcal{A}} = \mathcal{A}_1 \times \dots \times \mathcal{A}_N$, the joint observation space $\vec{\mathcal{O}} = \mathcal{O}_1 \times \dots \times \mathcal{O}_N$, a state transition function $\mathcal{T} : \mathcal{S} \times \vec{\mathcal{A}} \times \mathcal{S} \rightarrow [0, 1]$, the joint observation function $\mathcal{Z} : \mathcal{S} \times \vec{\mathcal{A}} \times \vec{\mathcal{O}} \rightarrow [0, 1]$, and a reward function $\mathcal{R} : \mathcal{S} \times \vec{\mathcal{A}} \rightarrow \mathbb{R}^N$.

At each state $s \in \mathcal{S}$, each agent $i \in \mathcal{I}$ simultaneously performs an action $a_i \in \mathcal{A}_i$, following which the environment transitions to the next state s' according to $\mathcal{T}$, and the agents receive an observation $o_i \in \mathcal{O}_i$, according to $\mathcal{Z}$, and reward $r_i \in \mathbb{R}$, according to $\mathcal{R}$. Each agent has no direct access to the environment state or knowledge of the other agent's actions and observations. Instead they must rely only on information in their *action-observation history* up to the current time step t: $h_{i,t} = \langle o_{i,0} a_{i,0} o_{i,1} \dots o_{i,t-1} a_{i,t-1} o_{i,t} \rangle$. Agents select their next action using their *policy* π_i which is a mapping from their history h_i (or belief, see Section 2.2) to a probability distribution over their actions. The goal of each agent i is to maximize its total expected *return* given by $J_i = \mathbb{E}\left[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{i,t'}\right]$, where $\gamma \in [0, 1)$ is a discount factor. Importantly, each agent's reward at each step also depends on the actions taken by the other agents present in the environment. Throughout this paper, we use i to denote the planning agent, $j \neq i$ to denote a single other agent, and $-i = \mathcal{I} \setminus i$ to denote the set of all other agents.

2.2 Decision-Theoretic Planning

Decision-theoretic planning addresses optimal behavior in uncertain environments by using a dynamics model of the environment to explicitly model the uncertainty arising from partial observability and stochastic outcomes of actions. In the multi-agent setting, uncertainty about the other agents – their policy, actions, and internal state – must also be considered (Seuken and Zilberstein 2008; Albrecht and Stone 2018). The various methods differ along a number of axes. However, a key axis is regards to the solution, specifically whether the solution is a joint policy for all agents (i.e. *centralized planning for decentralized control*) (Oliehoek and Amato 2016), or instead a policy for a single agent operating amongst independent other agents (Gmytrasiewicz and Doshi 2005). The POSG framework, and thus POSGGym, accommodates research in both directions. However, in this work, we concentrate on the latter: operating a single agent amongst independent co-players. In cooperative environments this is analogous to the problem of *ad-hoc teamwork* (Stone et al. 2010).

At the core of decision-theoretic planning lies an agent's *belief*, representing a distribution over possible states the agent could be in. In POMDPs, the belief b is a distribution over environment states $b \in \Delta(\mathcal{S})$. In the multi-agent setting, the policies and internal states of the other agents must also be considered. For the purpose of this work, a belief $b_{i,t} \in \Delta(\mathcal{S} \times \Pi_{-i} \times \mathcal{H}_{-i,t})$ encompasses the environment state $s \in S$, other agents' policies $\pi_{-i} \in \Pi_{-i}$, and their histories $h_{-i,t} \in \mathcal{H}_{-i,t}$, where the other agents' policy space is a finite discrete set. The planning agent uses its belief $b_{i,t}$ to select its next action according to its policy π_i and then updates its belief given its next observation.

2.3 Reinforcement Learning

Reinforcement Learning (RL) (Sutton and Barto 2018) is another approach for solving sequential-decision making problems. Whereas planning uses a dynamics model and explicit beliefs, RL focuses on learning a policy through interactions with the environment or through repeated simulations using a model. Deep RL in partially observable environments typically circumvent explicit beliefs by using recurrent networks (RNNs) (Bengio, Simard, and Frasconi 1994) such as LSTMs (Hochreiter and Schmidhuber 1997) to represent the policy (Hausknecht and Stone 2015). These RNNs can learn implicit beliefs, in that they learn representations of the sufficient statistics of the state of the world given the agents action-observation history. However, these learned implicit beliefs do not permit planning since search in planning requires sampling from beliefs in order to simulate possible future trajectories. Instead, prior work has combined RL with search by using a deep RL policy as the *search policy* within Monte Carlo Tree Search (MCTS) (Silver et al. 2018; Brown and Sandholm 2018; Lerer et al. 2020).

3 Related Work

3.1 Multi-Agent Libraries

Recent years have seen a proliferation of MARL research libraries. This includes general environment suites providing a standard API and a large collection of environments, such as PettingZoo (Terry et al. 2021), Melting Pot (Leibo et al. 2021) and JaxMARL (Rutherford et al. 2023). There are libraries focused on specific domains such as the StarCraft Multi-Agent Challenge (Samvelyan et al. 2019; Ellis et al. 2022), massively multi-agent online games (Suarez et al. 2019), environments with hundreds to millions of agents (Zheng et al. 2018), drones (Lechner et al. 2023), autonomous driving (Lopez et al. 2018; Zhang et al. 2019) and robotics (Tunyasuvunakool et al. 2020; Peng et al. 2021; Papoudakis et al. 2021). While these libraries offer a range of benchmarks they are designed specifically for MARL and so have no or limited model support and/or focus on limited domains, making them hard to use for general planning.

A number of libraries focusing on turn-based games have also been developed, including OpenSpiel (Lanctot et al. 2019), rlcard for card-games (Zha et al. 2019), and Pgx for accelerator-supported board games (Koyamada et al. 2024). Unlike most MARL libraries, these libraries support search by exposing the environment model in their APIs. However, each library models environments as Extensive Form Games (Osborne and Rubinstein 1994) which are better suited for strictly turn-based games, such as card and board games.

MADPToolbox (Spaan and Oliehoek 2008) is a collection of planning algorithms and benchmarks domains with a specific focus on discrete Dec-POMDPs. However, it is no longer maintained and its design makes it difficult to integrate with the modern library ecosystem. AdLeap-MAS (do Carmo Alves et al. 2022) also provides some support for planning, however it includes a very limited set of environments. More recently, libraries based on planning domain language description (PDDL) have been proposed, including PDDLGym (Silver and Chitnis 2020) and pyRDDLGym (Taitler et al. 2023). Like POSGGym, these libraries support both planning and learning, but primarily focus on single agent domains using the Gym API. In comparison, POSGGym includes a larger collection of multi-agent benchmarks, reference policies, and a multi-agent focused API design.

3.2 Multi-Agent Planning

In this work we empirically evaluate existing state-of-the-art methods for decision-theoretic planning in large partially observable, multi-agent environments. A closely related problem is that of *ad-hoc teamwork* (Stone et al. 2010), where methods based on stage games (Wu, Zilberstein, and Chen 2011), Bayesian beliefs (Barrett, Stone, and Kraus 2011; Barrett et al. 2014), types with parameters (Albrecht and Stone 2017), and for the many agent setting (Yourdshahi et al. 2018) have been proposed. All these methods use MCTS but are limited to environments where the state and actions of the other agents are fully observed. *POT-MMCP* (Schwartz, Kurniawati, and Hutter 2023) uses a meta-policy for guiding search and was shown to outperform related methods across a range of cooperative, competitive, and mixed environments in the type-based reasoning setting.

Several Monte Carlo planning methods have been proposed for solving large I-POMDPs. This includes methods based on finite-state automata (Panella and Gmytrasiewicz 2017) and for systems with communication (Kakarlapudi et al. 2022). However, most relevant for our setting are *IPOMCP* (Eck et al. 2020) and *INTMCP* (Schwartz, Zhou, and Kurniawati 2022). These two methods represent the current state-of-the-art for large, general I-POMDPs.

3.3 Combined Planning and Learning

Combining search with RL has been an important part of superhuman performance in games. Self-play RL and MCTS have been combined to achieve beyond expert performance in two-player fully-observable zero-sum games with both a known (Silver et al. 2016, 2018) and learned (Schrittwieser et al. 2020) environment model. Similar methods have been applied to zero-sum imperfect-information games (Brown and Sandholm 2018, 2019; Brown et al. 2020; Timbers et al. 2022), as well as cooperative games where there is prior coordination for decentralized execution (Lerer et al. 2020; Hu et al. 2021). Methods combining MCTS and RL for training a best-response policy against a distribution over policies (Li et al. 2023) have also been proposed. A key focus of existing work in combining RL and search has been for games which are fully-observable, where the agent has access to exact *information-states*, or using belief representations trained using specific domain knowledge. To the best of the authors knowledge ours is the first work to do a comprehensive empirical investigation combining RL with existing Monte Carlo planning methods based on particle filters.

4 POSGGym

The aim of POSGGym is to streamline planning and learning research in POSGs, with a particular emphasis on planning, since this is currently lacking in existing libraries. To accomplish this, POSGGym uses a general yet user-friendly API, and includes a diverse collection of environments and reference agents. POSGGym's API is based on the Gymnasium (Foundation 2022) and PettingZoo (Terry et al. 2021) libraries, as these are widely used by the RL community, making it simple to integrate POSGGym with the many MARL algorithm libraries compatible with PettingZoo.

4.1 API Design

The POSGGym API has three main components: *environment*, *model*, and *agents*.

```
import posggym
import posggym.agents as pga
env = posggym.make("PursuitEvasion-v1", grid="16x16")

policies = {
    "0": pga.make(
        "PursuitEvasion-v1/grid=16x16/RL1_i0-v0", env.model, "0"
    ),
    "1": pga.make(
        "PursuitEvasion-v1/ShortestPath-v0", env.model, "1"
    ),
}
seed = 42
obs, infos = env.reset(seed=seed)
for policy in policies.values():
    seed += 1
    policy.reset(seed=seed)

for _ in range(1000):
    actions = {i: policies[i].step(obs[i]) for i in env.agents}
    obs, rews, terms, truncs, all_done, infos = env.step(actions)

    if all_done:
        obs, infos = env.reset()
        for policy in policies.values():
            policy.reset()

env.close()
for policy in policies.values():
    policy.close()
```

Figure 2: POSGGym Environment and Agent APIs.

Environment API The environment API, depicted in Figure 2, closely follows the structure of PettingZoo's parallel environment API, however, with certain aspects aligning more closely with the Gymnasium API (see Appendix A.1 for a side-by-side comparison). At each timestep, each agent provides an `action`, which collectively form a joint action passed to the `step` function. This function updates the state of the environment and returns `observations`, `rewards`, `terminations`, `truncations`, `all_done`, `infos`. Each of these return values (except `all_done`) is a mapping from the ID of the agent to their respective return value. The `step` function mirrors PettingZoo's step function, with the addition of `all_done`, which indicates when all agents in the environment have reached a terminal state. Similarly for the `reset` method which resets the environment to a starting state and returns an `observation` and `info` for each active agent. The `render` and `close` function operate identically to Gymnasium: `render` provides a visual representation of the current state of the environment, while `close` shuts down the environment and performs any necessary clean-up.

Model API The model API provides access to a generative model of the environment for planning and also serves as the distinguishing feature between POSGGym and existing libraries. It closely resembles the environment API, except that models are stateless (apart from their random seed) and so the model methods take the environment `state` as an additional argument. More details and an example of the model API are provided in Appendix A.2.

By separating out the underlying model from the environment, agents can have access to the model without directly accessing the environment and its internal state (except through their actions of course). This makes it explicit which model functions can be used for planning (the model API), and what functions control the true environment (the environment API). It also allows for greater flexibility. For example, to study planning with inaccurate models one can provide agents with models that are different from the environment, say with simplified dynamics.

Agent API The goal of the Agent API is to offer a diverse collection of high-quality *reference policies* that can be leveraged for research. As we demonstrate in Section 5, the policies can be used to measure generalization by holding-out the policies for evaluation only, with no use of the policies during RL training or within the planning algorithm. Alternatively, the policies can be used for developing planning algorithms by using the policies to guide planning.

POSGGym's Agent API, depicted in Figure 2, follows a similar design to the Environment API. Its key methods include `make`, `step`, and `reset`. The `make` function initializes a new instance of a policy, from the policy's unique ID, the environment model, and the ID of the agent for which the policy will be used. It returns an instance of the `POSGGym.agents.Policy` class, the main Agent API class. The two main methods of the policy class are: `reset` which resets the policy to its initial state, and `step` which updates the policy with the agent's latest observation and returns the agent's next action. To support using reference policies during planning, the `POSGGym.agents.Policy` class includes additional methods that provide finer-grained control over the policy and its internal state, including methods for getting and setting the policy's internal state so it can be flexibly queried during planning. More details are provided in Appendix A.3.

4.2 Environments

POSGGym currently offers a collection of 14 environments which have been used in multi-agent research in various forms, with most existing only in paper descriptions, across disparate programming languages, some in unmaintained research code, and others in MARL libraries with no model support. Details and references for each environment currently supported by POSGGym are provided in Appendix B. For precise explanations of each environment, we refer the reader to POSGGym's documentation.

4.3 Reference Agents

POSGGym currently offers reference agents for the majority of its environments. These agents encompass a combination of handcrafted heuristic policies and policies trained using various MARL algorithms. For detailed information on the training methods for each policy, please refer to Appendix C. To access the comprehensive and up-to-date list of available policies, consult the library documentation.

5 Experiments

We used POSGGym to empirically evaluate planning, RL, and combined planning plus RL methods across diverse environments. The goal of our experiments was to gain novel insights into current state-of-the-art planning under uncertainty methods and compare these methods with integrated RL and planning. Our experiments serve an additional purpose of providing baseline results and algorithm implementations for POSGGym to facilitate future research[1].

5.1 Experiment Setup

In our experiments, we examine a planning agent interacting within a partially observable environment alongside one other agent with unknown behavior. We focus on the *type-based reasoning* setting (Albrecht, Crandall, and Ramamoorthy 2016) where the planning agent reasons about the policy or "type" of the other agent from a set of possible policies P. The goal of the planning agent is to maximize its expected reward given its uncertainty over the behaviour of the other agent and the state of the environment. We make no assumption about the reward structure of the environment and test across competitive, cooperative, and mixed domains.

For each environment the planning agent has access to a known population of policies P_{plan} for planning, and is evaluated against a test population P_{test}. We experiment with both the *in-distribution* setting where the test and planning population are the same ($P_{\text{plan}} = P_{\text{test}}$), and the *out-of-distribution* setting where they are different ($P_{\text{plan}} \neq P_{\text{test}}$). By comparing in- and out-of-distribution performance, we investigate generalization to novel co-players.

Two distinct populations, P_0 and P_1, are used for both planning P_{plan} and test P_{test} populations, with each experiment iterating over all planning and test population combinations, resulting in four distinct trials per algorithm, per environment. A uniform distribution over the policies within each population is used as the prior, so each policy had equal likelihood of being used by the other agent. For each environment, 10 to 12 policies were generated and split approximately evenly between P_0 and P_1, so that each contained either five or six policies. Further details on the policies are provided in Appendix C and Appendix D.

5.2 Planning Methods

We focus on planning methods designed for controlling a single agent within a large, partially observable, multi-agent environment. This limits us to online planning methods, and also excludes methods designed for environments with specific structures, e.g. with communication (Kakarlapudi et al. 2022), centralized control (Amato and Oliehoek 2014; Choudhury et al. 2022; Czechowski and Oliehoek 2021) or full observability (Yourdshahi et al. 2018). However, we highlight that POSGGym's general multi-agent API can be used for any planning method that is applicable to POSGs and its sub-classes, e.g. Dec-POMDPs, etc.

We compare the following planning methods:

POMCP (Silver and Veness 2010): Models the environment as a POMDP with the other agent's actions selected uniformly at random. This approach acts as a baseline with naive modeling for the other agent, UCB (Auer, Cesa-Bianchi, and Fischer 2002) for the exploration policy and Monte Carlo rollouts for node evaluation.

INTMCP (Schwartz, Zhou, and Kurniawati 2022): Models the environment as an I-POMDP and uses nested-MCTS to solve the I-POMDP at each reasoning level online. We use a reasoning level of $l = 2$ for our experiments, UCB for the exploration policy, and Monte Carlo rollouts for node evaluation. It does not incorporate P_{plan} into its decision-making, instead using a recursive I-POMDP model for the other agent.

IPOMCP (Eck et al. 2020; Kakarlapudi et al. 2022): Models the environment as an I-POMDP with uncertainty over the other agent's internal state (history and policy). Uses P_{plan} for modelling the other agent[2], UCB for the exploration policy and Monte Carlo rollouts for node evaluation.

POTMMCP (Schwartz, Kurniawati, and Hutter 2023): Similar to IPOMCP except the set of other agent policies are additionally utilized to inform planning via a meta-policy. Uses PUCB (Rosin 2011) for the exploration policy with the meta-policy as the search policy and Monte Carlo rollouts and pre-computed value function for node evaluations, depending on the policies in P_{plan}.

Table 1 shows a comparison of the key differences between each planning algorithm. For INTMCP and POMCP we only consider out-of-distribution performance as neither algorithm incorporates the planning population into their decision-making. Our experiments help evaluate how beneficial the inductive biases of these two methods are.

Table 1: Comparison of the planning and combined algorithms used in our experiments.

Algorithm	Search Policy	Explore Policy	Other Agent Model
INTMCP	Random	UCB	Nested MCTS
IPOMCP	Random	UCB	P_{plan}
POMCP	Random	UCB	Random
POTMMCP	Meta-Policy over P_{plan}	PUCB	P_{plan}
COMBINED	π_{BR}	PUCB	P_{plan}

For our experiments, we tested each method across a range of planning budgets $S \in [0.1, 1, 5, 10, 20]$, where S represents seconds of search time per step. Further details on each method and their hyperparameters are provided in Appendix E.

[1]Algorithm implementations and experiment code is available at: https://github.com/RDLLab/posggym-baselines

[2]In the original IPOMCP (Eck et al. 2020) and CIPOMCP (Kakarlapudi et al. 2022) papers a single policy for the other agent policy was generated using an I-POMDP solver; here we use the policies from the planning population.

5.3 Learning Method

For the learning method, we train a single deep RL policy as a best-response against each population, P_0 and P_1. We refer to the learning method as *Reinforcement Learning-Best Response* (RL-BR). To train each policy we use Proximal Policy Optimization (PPO) (Schulman et al. 2017) as the specific RL algorithm because it is used extensively for MARL research (Berner et al. 2019; Baker et al. 2020; Yu et al. 2022) and worked well across all environments we tested. In each environment, a separate RL-BR policy $\pi_{BR,k}$ was trained for each population $P_k \in \{P_0, P_1\}$. To ensure reproducibility in our results, we trained five versions of the same policy using different random seeds for each planning population in each environment. Each individual RL-BR policy was trained until convergence as indicated by their learning curve. The learning curves for each policy and training hyperparameters are provided in Appendix F.

5.4 Combined Planning and Learning Method

The combined planning and learning method incorporates the RL-BR policies from the previous section as a search policy within an MCTS based planner. POTMMCP (Schwartz, Kurniawati, and Hutter 2023) was used as the base planner with the search policy (normally the meta-policy) replaced with the RL-BR policy. We refer to this method simply as *Combined*. Table 1 compares its properties against the other planning methods. For the Combined method experiments we follow the same protocol used for the planning methods, testing across a range of planning budgets $S \in [0.1, 1, 5, 10, 20]$. We also repeat each experiment using each of the five RL-BR policies trained for each environment and planning population.

5.5 Experiment Environments

We tested each method on a range of cooperative, competitive, and mixed environments from POSGGym. This included *CooperativeReaching* (CR) (Rahman et al. 2023), *PredatorPrey* (Tan 1993; Leibo et al. 2017), *Driving* (McKee et al. 2022; Lerer and Peysakhovich 2019), *Level-Based Foraging* (LBF) (Christianos, Schäfer, and Albrecht 2020; Papoudakis et al. 2021), and *PursuitEvasion* (Seaman, van de Meent, and Wingate 2018; Schwartz, Zhou, and Kurniawati 2022). These environments have all been previously studied in the RL and planning literature and tested a range multi-agent concepts (Table 4 in the appendix). We limited the selection to those with discrete actions and observations, as this was what the planning methods in our experiments were designed for. The size of the various components of each environment are shown in Table 2.

5.6 Overall Results

We start by looking at the key high-level findings when averaging performance across the entire set of environments, shown in Figure 3. Averaging across environments allows us to look at performance across a broad distribution. To ensure all environments are weighted equally, we normalize the returns to be within $[0, 1]$, so that 0 and 1 correspond to the minimum and maximum possible return within each environment.

Table 2: Environment properties. $|S_0|$ denotes the number of possible initial states given the agent's initial observation.

Environment	$\lvert\mathcal{S}\rvert$	$\lvert S_0\rvert$	$\lvert\mathcal{A}_i\rvert$	$\lvert\mathcal{O}_i\rvert$
CR	625	1	5	625
Driving	8.9×10^{12}	9	5	6.6×10^{6}
LBF	1.5×10^{11}	5×10^{6}	6	30
PredatorPrey	7.2×10^{10}	64	5	100
PursuitEvasion i0	2.8×10^{8}	3	4	768
PursuitEvasion i1	2.8×10^{8}	1	4	4608

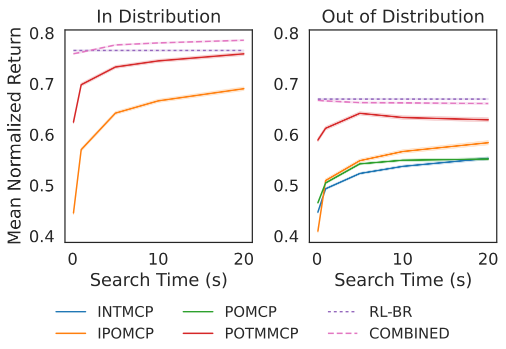


Figure 3: In-distribution (left) and out-of-distribution (right) performance of planning, learning, and combined methods averaged across all environments. Each plot shows the mean normalized return across search budgets (x-axis). Shaded areas show 95% confidence intervals

For the in-distribution setting, combining planning and learning improves on either approach alone, given enough planning time. We observe the benefits of incorporating planning alongside a trained RL policy when provided with an accurate model of the world and other agents. This finding aligns with previous research on combining search and RL games (Silver et al. 2016, 2018; Brown et al. 2020; Lerer et al. 2020; Hu et al. 2021). Unlike previous studies that utilize exact or learned belief models with factored public and private observations, here we show that combining a RL policy with a planner is also an effective method when using particle based beliefs. We believe this is a promising result for efforts to scale up planning to more complex domains, showing that existing particle based planning methods can benefit from an RL trained policy, and vice versa.

However, the relative gain in performance does not occur in the out-of-distribution setting where the model of the other agent is inaccurate. In fact, performance appears to degrade slightly with increased search time. In subsequent sections we discuss the possible explanations for this.

The RL-BR generally outperforms all pure planning methods in both in- and out-of-distribution settings. This demonstrates a general benefit of RL over pure planning within our experimental setup. However, the RL performance does not come without some cost, requiring a larger

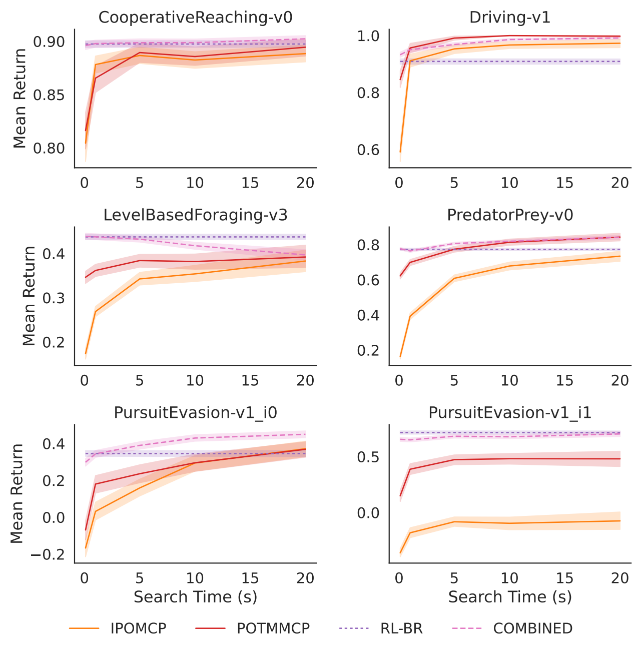


Figure 4: In-distribution performance of planning, learning, and combined methods in each environment. The plots show the mean return of each across planning budgets (x-axis). Shaded areas show 95% confidence intervals

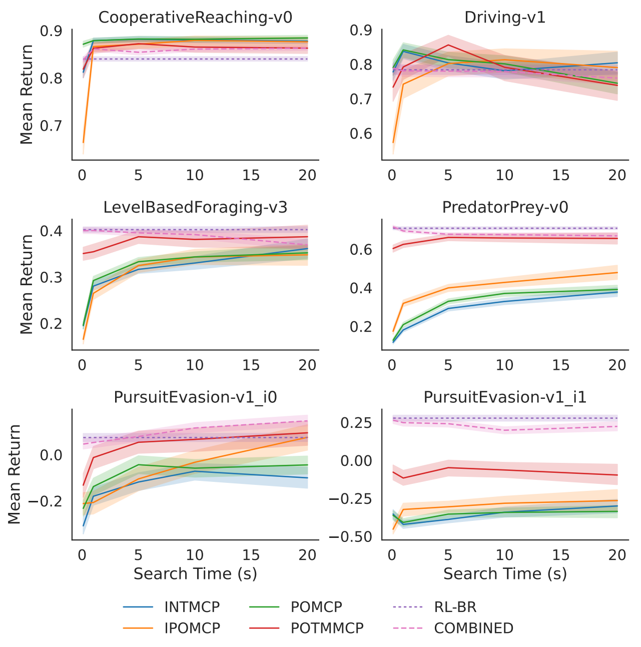


Figure 5: Out-of-distribution performance of planning, learning, and combined methods in each environment. The plots show the mean return across planning budgets (x-axis). Shaded areas show 95% confidence intervals

amount of offline compute for training –up to 48 hours on 32 CPUs and 1 GPU per policy– compared to the online planners. Nonetheless, RL policies offer faster per-step execution time. Our observations highlight a key advantage of modern deep RL based methods, namely their ability to effectively leverage large amounts of compute.

For the in-distribution setting, planning methods exhibit improved performance with longer search time, with POTMMCP even matching RL-BR's performance with 20 s of search time per step. We observe that planning methods tend to converge towards optimality in environments where they posses a perfect model of the environment and the other agent. Notably, in some specific environments, planning methods outperform RL-BR, as shown in Figure 4. We discuss this further in Section 5.7.

Monotonic improvement in performance with search time is evident for all planning and combined methods in the in-distribution setting, contrasting with the out-of-distribution setting. While accurate models of the environment and the other agent lead to performance gain with increased search time, this trend does not hold uniformly in multi-agent settings. Notably, when the test population differs from the planning population, we see performance plateau or even decline with planning budget. Further exploration into the underlying cause of this discrepancy is discussed in Section 5.8.

A large gap exists between in- and out-of-distribution performance across all methods. This contrast in performance (also shown in Appendix G) highlights the impact of inaccurate other agent models, regardless of whether methods are learning or planning based, or a combination of both, and indicates the general brittleness of the tested methods when encountering novel partners.

However, it's worth pointing out that our results are influenced by the specific planning and test populations used. Adjusting the planning population, such as a larger or more diverse population based on some diversity metric, could potentially narrow this performance gap. The design of populations to enhance the robustness of autonomous agents in multi-agent settings is an active area of research (Lanctot et al. 2017; Lupu et al. 2021; Xing et al. 2021; Rahman et al. 2023), and our finding emphasize its significance for both planning and RL.

5.7 In-Distribution Performance

Figure 4 shows in-distribution performance in each environment. Contrary to the overall results, we observe more nuanced trends at the individual environment level.

In some cases, combining planning and learning can lead to deteriorating performance as the planning budget increases. Specifically, in the LBF environment, we observed a decline in performance for the combined method as the search duration increased, contrary to expectations. The result highlights a limitation of current particle-based planners, namely the introduction of error due to poor belief approximation.

Specifically, the number of possible initial states in LBF ($|S_0| = 5 \times 10^6$), as shown in Table 2, is significantly larger than in any other environment. Conversely, the maximum number of particles used for initial belief representation in our experiments was 2320, which, although more than double the amount used in prior research (Eck et al. 2020; Kakarlapudi et al. 2022), is still a minor fraction of LBF's initial state space. This discrepancy is exacerbated by the fact that many state features in LBF do not change post-initialization. Since all our planning methods rely on Bayesian updates for the beliefs, if the true value of a state features is not within the initial belief, the planner will never be aware of the true value, even with unlimited planning budget post-initial belief. This is evidenced by the low probability (less than 0.04) assigned to the true state by the combined agent's belief, as detailed in Sup. Figure 15. The belief inaccuracy causes the policy produced by planning to diverge from the optimal policy, and away from the RL policy as planning time increases.

In four of six tested environments, at least one planning method either matched or outperformed the RL-BR learning approach given adequate planning time. Specifically, POTMMCP excelled in the Driving and PredatorPrey environments and equaled performance in CR and PursuitEvasion_i0, contrasting with the overall results where RL-BR does better. One explanation for this is the planning methods' superior generalization to novel situations, given an accurate model. With an accurate belief and model of the environment, planning allows the agent to improve its action value estimates online for any encountered state, regardless of the state's rarity. In contrast, RL-BR performs all policy improvement offline, making it brittle when faced with situations during execution rarely encountered during training. Evidence of this can be seen in the Driving environment where all methods succeed the vast majority of the time ($> 93\%$), however in the rare failures RL-BR crashes significantly more often than even the worst performing planning method ($4.14\% vs 0.98\%$, Table 3). We hypothesize planning methods are able to improve on the robustness of RL-BR in a minority of situations, likely those rare in RL-BR's training distribution. When mistakes have large consequences, this can have a significant impact on final performance.

Table 3: Percentage of in-distribution *Driving* environment episodes that ended with a crash (*Crashed*), reaching the goal (*Success*), or the step limit being reached (*Timedout*). Results are using the maximum search time (20 s).

Algorithm	Crashed	Success	Timedout
IPOMCP	0.98%	97.46%	1.56%
POTMMCP	0.12%	99.88%	0.00%
RL-BR	4.14%	93.12%	2.74%
COMBINED	0.46%	99.51%	0.03%

Conversely, in the LBF and PursuitEvasion_i1 environments, no planning approach surpassed RL-BR. In LBF, the limitation stemmed from the belief inaccuracy discussed above. In PursuitEvasion_i1, the challenge lies in the long horizon planning needed in this environment in order to find the optimal policy of reaching the goal. RL-BR is able to leverage significantly more compute to handle the long horizons, actually requiring $10\times$ more training steps to converge in PursuitEvasion_i1 compared to PursuitEvasion_i0 (Sup. Figure 12)). The planning methods on the other hand have a limited search budget which in our experiments was not enough to plan to the necessary horizon. We see evidence of this in the longer episode lengths for planning methods versus RL-BR, with the gap in episode length between learning and planning methods much greater in PursuitEvasion_i1 than for any other environment (Sup. Figure 17). The longer episodes in this case indicate the agent prioritizing avoiding the other agent (the sub-optimal, short-horizon strategy), compared with going for the goal. Fortunately, improved search policies, such as used in the Combined method, offer a way to overcome the challenge of planning over long-horizons.

5.8 Out-of-Distribution Performance

In Figure 5 we show out-of-distribution performance within each environment.

Combining learning and planning leads to worse performance than learning alone in four of six environments. This outcome is largely attributed to poor belief accuracy similar to the LBF environment in the in-distribution setting. Incorrect models of the other agent in the out-of-distribution setting results in worse beliefs about the environment state compared to the in-distribution setting, despite having a perfect environment model in both cases (shown in Sup. Figure 15). Finding robust methods for modelling the other agents, and planning techniques that better handle model inaccuracy are both important directions for future work.

Planning methods that modeled the other agent naively (POMCP) or recursively (INTMCP) generally performed worse than other approaches. While frameworks based on recursive reasoning hold promise for domains like human-robot interaction (Woodward and Wood 2012), they showed limited utility in our experiments, where the other agent policies did not behave randomly or employ explicit recursive reasoning. Rather our experiments show the benefits that can be gained by explicitly considering the possible types of other agent behaviour.

6 Conclusion and Future Directions

In this paper, we presented POSGGym, a library facilitating research at the intersection of decision-theoretic planning and RL. Using POSGGym we conducted an extensive empirical evaluation of existing state-of-the-art planning under uncertainty methods and combined planning and RL. Our results demonstrate that combining RL with particle based planning can be effective in large, multi-agent environments. They also highlight important directions for future research, namely improving robustness to novel partners, and exploring scalable representations of complex beliefs. We hope these contributions will spur further research on the integration of planning and RL in partially observable multi-agent domains, so as to gain the best of both model-driven and data-driven techniques.

References

Albrecht, S.; Crandall, J.; and Ramamoorthy, S. 2016. Belief and Truth in Hypothesised Behaviours. *Artificial Intelligence*, 235: 63–94.

Albrecht, S.; and Stone, P. 2017. Reasoning about Hypothetical Agent Behaviours and Their Parameters. *AAMAS*.

Albrecht, S.; and Stone, P. 2018. Autonomous Agents Modelling Other Agents: A Comprehensive Survey and Open Problems. *Artificial Intelligence*, 258: 66–95.

Amato, C.; and Oliehoek, F. A. 2014. Scalable Planning and Learning for Multiagent POMDPs: Extended Version. *arxiv preprint arXiv:1404.1140*.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine learning*, 47(2): 235–256.

Baker, B.; Kanitscheider, I.; Markov, T.; Wu, Y.; Powell, G.; McGrew, B.; and Mordatch, I. 2020. Emergent Tool Use From Multi-Agent Autocurricula. *arxiv preprint arXiv:1909.07528*.

Barrett, S.; Agmon, N.; Hazon, N.; Kraus, S.; and Stone, P. 2014. Communicating with Unknown Teammates. In *ECAI*, 45–50.

Barrett, S.; Stone, P.; and Kraus, S. 2011. Empirical Evaluation of Ad Hoc Teamwork in the Pursuit Domain. In *Autonomous Agents and Multiagent Systems*, 567–574.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning Long-Term Dependencies with Gradient Descent Is Difficult. *IEEE transactions on neural networks*, 5(2): 157–166.

Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Debiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; and Hesse, C. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680*.

Bernstein, D.; Givan, R.; Immerman, N.; and Zilberstein, S. 2002. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4).

Blomqvist, V. 2023. Pymunk.

Blythe, J. 1999. Decision-Theoretic Planning. *AI magazine*, 20(2): 37–37.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 11: 1–94.

Brown, N.; Bakhtin, A.; Lerer, A.; and Gong, Q. 2020. Combining Deep Reinforcement Learning and Search for Imperfect-Information Games. *Advances in Neural Information Processing Systems*, 33: 17057–17069.

Brown, N.; and Sandholm, T. 2018. Superhuman AI for Heads-up No-Limit Poker: Libratus Beats Top Professionals. *Science*, 359(6374).

Brown, N.; and Sandholm, T. 2019. Superhuman AI for Multiplayer Poker. *Science*, 365(6456): 885–890.

Carr, S.; Jansen, N.; Bharadwaj, S.; Spaan, M. T.; and Topcu, U. 2021. Safe Policies for Factored Partially Observable Stochastic Games. In *Robotics: Science and Systems*.

Choudhury, S.; Gupta, J. K.; Morales, P.; and Kochenderfer, M. J. 2022. Scalable Online Planning for Multi-Agent MDPs. *Journal of Artificial Intelligence Research*, 73: 821–846.

Christianos, F.; Schäfer, L.; and Albrecht, S. V. 2020. Shared Experience Actor-Critic for Multi-Agent Reinforcement Learning. *NeurIPS*.

Cui, B.; Hu, H.; Pineda, L.; and Foerster, J. 2021. K-Level Reasoning for Zero-Shot Coordination in Hanabi. *NeurIPS*.

Czechowski, A.; and Oliehoek, F. A. 2021. Decentralized MCTS via Learned Teammate Models. In *International Joint Conferences on Artificial Intelligence*, 81–88.

De Souza, C.; Newbury, R.; Cosgun, A.; Castillo, P.; Vidolov, B.; and Kulić, D. 2021. Decentralized Multi-Agent Pursuit Using Deep Reinforcement Learning. *Robotics and Automation Letters*, 6(3).

de Witt, C.; Gupta, T.; Makoviichuk, D.; Makoviychuk, V.; Torr, P.; Sun, M.; and Whiteson, S. 2020. Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge? *arxiv preprint arXiv:2011.09533*.

do Carmo Alves, M. A.; Varma, A.; Elkhatib, Y.; and Soriano Marcolino, L. 2022. AdLeap-MAS: An Open-Source Multi-Agent Simulator for Ad-Hoc Reasoning. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, 1893–1895.

Eck, A.; Shah, M.; Doshi, P.; and Soh, L.-K. 2020. Scalable Decision-Theoretic Planning in Open and Typed Multiagent Systems. *AAAI*.

Ellis, B.; Moalla, S.; Samvelyan, M.; Sun, M.; Mahajan, A.; Foerster, J.; and Whiteson, S. 2022. SMACv2: An Improved Benchmark for Cooperative Multi-Agent Reinforcement Learning. *CoRR*, abs/2212.07489.

Foundation, F. 2022. Gymnasium.

Gmytrasiewicz, P.; and Doshi, P. 2005. A Framework for Sequential Planning in Multi-Agent Settings. *JAIR*, 24.

Gupta, J.; Egorov, M.; and Kochenderfer, M. 2017. Cooperative multi-agent control using deep reinforcement learning. *AAMAS*.

Hansen, E.; Bernstein, D.; and Zilberstein, S. 2004. Dynamic Programming for Partially Observable Stochastic Games. *National Conference on Artifical Intelligence*.

Hausknecht, M.; and Stone, P. 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. In *2015 AAAI Fall Symposium Series*.

Hochreiter, S.; and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural computation*, 9(8): 1735–1780.

Hu, H.; Lerer, A.; Brown, N.; and Foerster, J. 2021. Learned Belief Search: Efficiently Improving Policies in Partially Observable Settings. *arxiv preprint arXiv:2106.09086*.

Kaelbling, L.; Littman, M.; and Cassandra, A. 1998. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2): 99–134.

Kakarlapudi, A.; Anil, G.; Eck, A.; Doshi, P.; and Soh, L.-K. 2022. Decision-Theoretic Planning with Communication in Open Multiagent Systems. *UAI*.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

Koyamada, S.; Okano, S.; Nishimori, S.; Murata, Y.; Habara, K.; Kita, H.; and Ishii, S. 2024. Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning. *arxiv preprint arXiv:2303.17503*.

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

Lanctot, M.; Lockhart, E.; Lespiau, J.-B.; Zambaldi, V.; Upadhyay, S.; Pérolat, J.; Srinivasan, S.; Timbers, F.; Tuyls, K.; Omidshafiei, S.; Hennes, D.; Morrill, D.; Muller, P.; Ewalds, T.; Faulkner, R.; Kramár, J.; Vylder, B. D.; Saeta, B.; Bradbury, J.; Ding, D.; Borgeaud, S.; Lai, M.; Schrittwieser, J.; Anthony, T.; Hughes, E.; Danihelka, I.; and Ryan-Davis, J. 2019. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR*, abs/1908.09453.

Lanctot, M.; Zambaldi, V.; Gruslys, A.; Lazaridou, A.; Tuyls, K.; Pérolat, J.; Silver, D.; and Graepel, T. 2017. A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. *Advances in Neural Information Processing Systems*, 30.

Lechner, M.; Yin, L.; Seyde, T.; Wang, T.-H.; Xiao, W.; Hasani, R.; Rountree, J.; and Rus, D. 2023. Gigastep - One Billion Steps per Second Multi-agent Reinforcement Learning. In *Thirty-Seventh Conferences on Neural Information Processing Systems Datasets and Benchmarks Track*.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep Learning. *nature*, 521(7553).

Leibo, J.; Dueñez-Guzman, E.; Vezhnevets, A.; Agapiou, J.; Sunehag, P.; Koster, R.; Matyas, J.; Beattie, C.; Mordatch, I.; and Graepel, T. 2021. Scalable Evaluation of Multi-Agent Reinforcement Learning with Melting Pot. *ICML*.

Leibo, J.; Zambaldi, V.; Lanctot, M.; Marecki, J.; and Graepel, T. 2017. Multi-Agent Reinforcement Learning in Sequential Social Dilemmas. *AAMAS*.

Lerer, A.; Hu, H.; Foerster, J.; and Brown, N. 2020. Improving Policies via Search in Cooperative Partially Observable Games. *AAAI Conference on Artificial Intelligence*, 34(05): 7187–7194.

Lerer, A.; and Peysakhovich, A. 2019. Learning existing social conventions via observationally augmented self-play. *AIES*.

Li, Z.; Lanctot, M.; McKee, K. R.; Marris, L.; Gemp, I.; Hennes, D.; Muller, P.; Larson, K.; Bachrach, Y.; and Wellman, M. P. 2023. Combining Tree-Search, Generative Models, and Nash Bargaining Concepts in Game-Theoretic Reinforcement Learning. *arxiv preprint arXiv:2302.00797*.

Lopez, P. A.; Behrisch, M.; Bieker-Walz, L.; Erdmann, J.; Flötteröd, Y.-P.; Hilbrich, R.; Lücken, L.; Rummel, J.; Wagner, P.; and Wießner, E. 2018. Microscopic Traffic Simulation Using Sumo. *ITSC*.

Lupu, A.; Cui, B.; Hu, H.; and Foerster, J. 2021. Trajectory Diversity for Zero-Shot Coordination. In *International Conference on Machine Learning*, 7204–7213. PMLR.

McKee, K.; Leibo, J.; Beattie, C.; and Everett, R. 2022. Quantifying the effects of environment and population diversity in multi-agent reinforcement learning. *AAMAS*.

Ng, B.; Meyers, C.; Boakye, K.; and Nitao, J. 2010. Towards Applying Interactive POMDPs to Real-World Adversary Modeling. In *Innovative Applications of Artificial Intelligence*.

Oliehoek, F. A.; and Amato, C. 2016. *A Concise Introduction to Decentralized POMDPs*. SpringerBriefs in Intelligent Systems. Springer International Publishing.

Ooi, J.; and Wornell, G. 1996. Decentralized Control of a Multiple Access Broadcast Channel: Performance Bounds. *Conference on Decision and Control*, 1.

Osborne, M.; and Rubinstein, A. 1994. *A Course in Game Theory*. MIT press.

Panella, A.; and Gmytrasiewicz, P. 2017. Interactive POMDPs with Finite-State Models of Other Agents. *AAMAS*.

Papoudakis, G.; Christianos, F.; Schäfer, L.; and Albrecht, S. 2021. Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks. *NeurIPS Track on Datasets and Benchmarks*.

Peng, B.; Rashid, T.; Schroeder de Witt, C.; Kamienny, P.-A.; Torr, P.; Böhmer, W.; and Whiteson, S. 2021. FACMAC: Factored Multi-Agent Centralised Policy Gradients. *NeurIPS*.

Rahman, A.; Fosong, E.; Carlucho, I.; and Albrecht, S. 2023. Generating Teammates for Training Robust Ad Hoc Teamwork Agents via Best-Response Diversity. *arxiv preprint arXiv:2207.14138*.

Rosin, C. D. 2011. Multi-Armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence*, 61(3): 203–230.

Rutherford, A.; Ellis, B.; Gallici, M.; Cook, J.; Lupu, A.; Ingvarsson, G.; Willi, T.; Khan, A.; de Witt, C. S.; Souly, A.; Bandyopadhyay, S.; Samvelyan, M.; Jiang, M.; Lange, R. T.; Whiteson, S.; Lacerda, B.; Hawes, N.; Rocktäschel, T.; Lu, C.; and Foerster, J. N. 2023. JaxMARL: Multi-Agent RL Environments in JAX. In *Second Agent Learning in Open-Endedness Workshop*.

Samvelyan, M.; Rashid, T.; de Witt, C. S.; Farquhar, G.; Nardelli, N.; Rudner, T.; Hung, C.-M.; Torr, P.; Foerster, J.; and Whiteson, S. 2019. The StarCraft Multi-Agent Challenge. *AAMAS*.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; and Graepel, T. 2020. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839): 604–609.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.

Schwartz, J.; Kurniawati, H.; and Hutter, M. 2023. Combining a Meta-Policy and Monte-Carlo Planning for Scalable Type-Based Reasoning in Partially Observable Environments. *arXiv preprint arXiv:2306.06067*.

Schwartz, J.; Zhou, R.; and Kurniawati, H. 2022. Online Planning for Interactive-POMDPs Using Nested Monte Carlo Tree Search. *IROS*.

Seaman, I. R.; van de Meent, J.-W.; and Wingate, D. 2018. Nested Reasoning About Autonomous Agents Using Probabilistic Programs. *arXiv preprint arXiv:1812.01569*.

Seuken, S.; and Zilberstein, S. 2008. Formal Models and Algorithms for Decentralized Decision Making under Uncertainty. *Autonomous Agents and Multi-Agent Systems*, 17: 190–250.

Seymour, R.; and Peterson, G. L. 2009. A Trust-Based Multiagent System. In *International Conference on Computational Science and Engineering*, volume 3, 109–116. IEEE.

Shapley, L. 1953. Stochastic Games. *Proceedings of the National Academy of Sciences*, 39(10).

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; and Lanctot, M. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *nature*, 529(7587).

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; and Graepel, T. 2018. A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play. *Science*, 362(6419): 1140–1144.

Silver, D.; and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. *NeurIPS*.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *International Conference on Automated Planning and Scheduling PRL Workshop*.

Spaan, M.; and Oliehoek, F. 2008. The MultiAgent Decision Process Toolbox: Software for Decision-Theoretic Planning in Multiagent Systems. *Proceedings of the AAMAS Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains (MSDM)*.

Stone, P.; Kaminka, G. A.; Kraus, S.; and Rosenschein, J. S. 2010. Ad Hoc Autonomous Agent Teams: Collaboration without Pre-Coordination. In *AAAI*.

Suarez, J.; Du, Y.; Isola, P.; and Mordatch, I. 2019. Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents. *arXiv preprint arXiv:1903.00784*.

Sunberg, Z.; and Kochenderfer, M. 2018. Online Algorithms for POMDPs with Continuous State, Action, and Observation Spaces. *International Conference on Automated Planning and Scheduling*, 28: 259–263.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Taitler, A.; Gimelfarb, M.; Jeong, J.; Gopalakrishnan, S.; Mladenov, M.; Liu, X.; and Sanner, S. 2023. pyRDDLGym: From RDDL to Gym Environments. In *International Conference on Automated Planning and Scheduling PRL Workshop*.

Tan, M. 1993. Multi-Agent Reinforcement Learning: Independent vs Cooperative Agents. *ICML*.

Terry, J.; Black, B.; Grammel, N.; Jayakumar, M.; Hari, A.; Sullivan, R.; Santos, L.; Dieffendahl, C.; Horsch, C.; Perez-Vicente, R.; et al. 2021. Pettingzoo: Gym for Multi-Agent Reinforcement Learning. *NeurIPS*.

Tesauro, G. 1994. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural computation*, 6(2).

Timbers, F.; Bard, N.; Lockhart, E.; Lanctot, M.; Schmid, M.; Burch, N.; Schrittwieser, J.; Hubert, T.; and Bowling, M. 2022. Approximate Exploitability: Learning a Best Response. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 3487–3493.

Tunyasuvunakool, S.; Muldal, A.; Doron, Y.; Liu, S.; Bohez, S.; Merel, J.; Erez, T.; Lillicrap, T.; Heess, N.; and Tassa, Y. 2020. Dm_control: Software and Tasks for Continuous Control. *Software Impacts*, 6.

Woodward, M. P.; and Wood, R. J. 2012. Learning from Humans as an I-POMDP. *arXiv preprint arXiv:1204.0274*.

Wu, F.; Zilberstein, S.; and Chen, X. 2011. Online Planning for Ad Hoc Autonomous Agent Teams. In *International Joint Conference on Artificial Intelligence*.

Xing, D.; Liu, Q.; Zheng, Q.; Pan, G.; and Zhou, Z. H. 2021. Learning with Generated Teammates to Achieve Type-Free Ad-Hoc Teamwork. In *IJCAI*, 472–478.

Yourdshahi, E. S.; Pinder, T.; Dhawan, G.; Marcolino, L. S.; and Angelov, P. 2018. Towards Large Scale Ad-Hoc Teamwork. In *International Conference on Agents*, 44–49. IEEE.

Yu, C.; Velu, A.; Vinitsky, E.; Gao, J.; Wang, Y.; Bayen, A.; and Wu, Y. 2022. The Surprising Effectiveness of Ppo in Cooperative Multi-Agent Games. *Advances in Neural Information Processing Systems*, 35: 24611–24624.

Zha, D.; Lai, K.-H.; Cao, Y.; Huang, S.; Wei, R.; Guo, J.; and Hu, X. 2019. RLCard: A Toolkit for Reinforcement Learning in Card Games. *arxiv preprint arXiv:1910.04376*.

Zhang, H.; Feng, S.; Liu, C.; Ding, Y.; Zhu, Y.; Zhou, Z.; Zhang, W.; Yu, Y.; Jin, H.; and Li, Z. 2019. Cityflow: A Multi-Agent Reinforcement Learning Environment for Large Scale City Traffic Scenario. *The World Wide Web Conference*.

Zheng, L.; Yang, J.; Cai, H.; Zhou, M.; Zhang, W.; Wang, J.; and Yu, Y. 2018. Magent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence. *AAAI*, 32.

A POSGGym API

Here we provide some additional details about POSGGym's API not included in the main text. The high-level design for the main API is shown in Figure 6.

A.1 Environment API Comparison

The POSGGym environment API, depicted in Figure 2, closely follows the structure of PettingZoo's parallel environment API, however, with certain aspects aligning more closely with the Gymnasium API. Figure 7 shows a comparison between the three libraries.

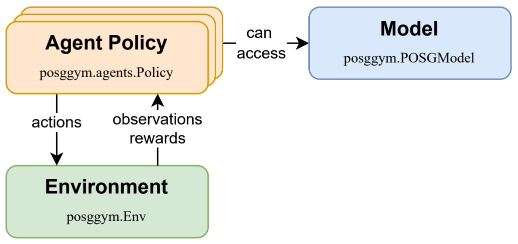


Figure 6: High-level design of POSGGym. Agents interact with the environment by selecting actions according to their policy which has access to a model of the environment for planning.

POSGGym's environment API differs from the PettingZoo parallel environment API in two key aspects: the `make` function for environment initialization, and the inclusion of `all_done` in the `step` method's return values. The use of the `make` function aligns more closely with the design of Gymnasium, offering users more convenience and control. While the inclusion of `all_done` differs from both PettingZoo and Gymnasium. This addition aims to simplify the tracking of agent termination during an episode, which can be non-trivial in open environments where the active agents may change over time. It also accounts for scenarios where agents can leave or join the environment within a single episode.

To make integration with existing MARL libraries easier, POSGGym provides a `PettingZoo` wrapper class that enables the conversion of any POSGGym environment into an equivalent PettingZoo parallel API environment. By using this wrapper, any POSGGym environment can be used seamlessly with any library that supports the PettingZoo API.

A.2 Model API

In Figure 8 we show POSGGym's Model API in action. The main model API methods are:

- `sample_initial_state` – samples an initial environment state
- `sample_initial_obs` – samples initial observations for each agent given a state
- `get_agents` – returns the IDs of agents that are active in a given state
- `step` – similar to the environment `step` function, but also returns the next state. It takes both a state and joint actions as arguments
- `seed` – sets the random seed for the model

Full POSG Model API In addition to the generative model functionality, POSGGym defines the `POSGGym.POSGFullModel` API, which extends the `POSGGym.POSGModel` class to include all components of the formal POSG definition. This extension allows POSGGym to be used for defining models for planning algorithms that require the full model, rather than just a generative model. Specifically, the `POSGGym.POSGFullModel` includes the following additional methods:

- `get_initial_belief` – returns the initial state distribution S_0
- `transition_fn` – defines the state transition function $\mathcal{T} : \mathcal{S} \times \vec{\mathcal{A}} \times \mathcal{S} \to [0, 1]$
- `observation_fn` – defines the joint observation function $\mathcal{Z} : \mathcal{S} \times \vec{\mathcal{A}} \times \vec{\mathcal{O}} \to [0, 1]$
- `reward_fn` – defines the joint reward function $\mathcal{R} : \mathcal{S} \times \vec{\mathcal{A}} \to \mathbb{R}^N$

The full model definition is not included in the main `POSGGym.POSGModel` model class due to the difficultly of implementing it in environments with very large state, action, or observation spaces and complex dynamics. In such cases, it is often more practical and common to define a generative model that can be used for sample-based planning approaches like MCTS.

```
import posggym
env = posggym.make("PursuitEvasion-v1", render_mode="human")
obs, infos = env.reset(seed=42)

for _ in range(1000):
    actions = {i: policies[i](obs[i]) for i in env.agents}
    obs, rews, terms, truncs, all_done, infos = env.step(actions)

    if all_done:
        obs, infos = env.reset()

env.close()
```

(a) POSGGym

```
from pettingzoo.butterfly import pistonball_v6
env = pistonball_v6.parallel_env(render_mode="human")
obs = env.reset(seed=42)

for _ in range(1000):
    actions = {i: policies[i](obs[i]) for i in env.agents}
    obs, rew, terms, truncs, infos = env.step(actions)

    if not env.agents:
        obs = env.reset()

env.close()
```

(b) PettingZoo

```
import gymnasium as gym
env = gym.make("LunarLander-v2", render_mode="human")
obs, info = env.reset(seed=42)

for _ in range(1000):
   action = policy(obs)
   obs, rew, term, trunc, info = env.step(action)

   if term or trunc:
      obs, info = env.reset()

env.close()
```

(c) Gymnasium

Figure 7: Comparison of POSGGym, Gymnasium, and PettingZoo Environment APIs

```
import posggym
env = posggym.make("PredatorPrey-v0")
model = env.model
model.seed(seed=42)

state = model.sample_initial_state()
obs = model.sample_initial_obs(state)

for t in range(50):
    actions = {i: policies[i](obs[i]) for i in model.get_agents(state)}
    timestep = model.step(state, actions)

    # timestep attribute can be accessed individually:
    state = timestep.state
    obs = timestep.observations

    # Or unpacked fully
    # state, obs, rews, terms, truncs, all_done, infos = timestep

    if timestep.all_done:
        state = model.sample_initial_state()
        obs = model.sample_initial_obs(state)
```

Figure 8: POSGGym Model API.

A.3 Agent API

When tackling POSGs, a critical consideration is the requirement for policies to maintain an internal state to handle the partial observability of the environment. Approaches to tackle this challenge vary: some utilize explicit beliefs where agents maintain and update a probabilistic representation of the unobservable features of the environment, while others rely on implicit beliefs that leverage learned representations or neural network architectures to capture relevant information from observations. Incorporating these crucial elements into decision-making processes allows policies to exhibit more sophisticated and adaptive behaviors within complex environments. To support internal states, the `POSGGym.agents.Policy` class also includes a number of additional methods that provide information and finer-grained control over the policy. These methods include:

- `get_initial_state` - returns the initial state of the policy. For example, the initial hidden state for a RNN based policy.
- `get_next_state` - returns the next policy state, given the current policy state and the next observation.
- `sample_action` - sample an action given a policy state
- `get_pi` - get the distribution over actions given a policy state
- `set_state` - set the internal state of the policy
- `get_state` - get the internal state of the policy
- `get_state_from_history` - unrolls the policy to get its state given an action-observation history.

All together, the API provides enough control that the policy can be used for evaluation using the `step` method, or for planning using the finer-grained control methods like `get_next_state` and `sample_action`.

B POSGGym Environments

POSGGym currently supports 14 different environments. Table 4 shows the complete list of environments, along with some of their properties and the multi-agent concepts they involve.

B.1 Classic

POSGGym includes several well-known problems that have been used extensively in planning and multi-agent research. These include Multi-Access Broadcast Channel (MABC) (Ooi and Wornell 1996; Hansen, Bernstein, and Zilberstein 2004), Multi-Agent Tiger (Gmytrasiewicz and Doshi 2005), and Rock-Paper-Scissors. These problems encompass co-operative, mixed, and competitive scenarios, respectively, and support discrete actions and observations. Due to their smaller size and well-defined characteristics, some versions of these problems have known provably optimal solutions. This makes them useful for debugging and for fine-grained analysis of algorithms. POSGGym offers full model definitions for all the classic problems currently implemented in the library.

B.2 Grid-World

Six widely used discrete grid-world problems are also provided by POSGGym. These problems encompass a variety of scenarios, including Cooperative Reaching (CR) (Rahman et al. 2023), Level Based Foraging (LBF) (Christianos, Schäfer, and Albrecht 2020; Papoudakis et al. 2021), Two Paths (Schwartz, Zhou, and Kurniawati 2022), Unmanned Aerial Vehicle (UAV) (Panella and Gmytrasiewicz 2017), Driving (McKee et al. 2022; Lerer and Peysakhovich 2019), Predatory Prey (Tan 1993; Leibo et al. 2017) and Pursuit Evasion (Seaman, van de Meent, and Wingate 2018; Schwartz, Zhou, and Kurniawati 2022). The current selection of problems was chosen to provide a diverse range of cooperative, mixed and competitive environments, as well as symmetric and asymmetric roles. Each environment is represented as a grid-world with discrete observations and actions.

B.3 Continuous

POSGGym also offers four 2D continuous problems, including extensions of grid-world environments (Driving, Predator Prey and Pursuit Evasion) which have been adapted to a continuous domain. The dynamics of the agents are modeled using a simple non-holonomic unicycle model, where agents are controlled by both angular and linear velocities. PyMunk (Blomqvist 2023) is employed as the physics engine to support these dynamics. The observation incorporate sensors that emit from the agents' positions in a circular pattern at a fixed distance. This approach aligns with the observation model used in PettingZoo's WaterWorld environment (Gupta, Egorov, and Kochenderfer 2017). The fourth environment is the Drone Team Capture (DTC) environment (De Souza et al. 2021), which simulates a cooperative pursuit-evasion scenario. POSGGym's implementation of DTC closely adheres to the original paper, with enhancements to accommodate partial observability, such as limited sight distance.

C Agent Populations

POSGGym comes with a diverse set of policies for the majority of the supported environments. Depending on the environment the set of policies will be made of up of a mix

Table 4: Properties and multi-agent concepts of POSGGym environments (related properties are grouped by row color).

		Classic			Grid-World							Continuous			
		MABC	MA Tiger	RPS	CR	LBF	Two Paths	UAV	Driving	Predator Prey	Pursuit Evasion	Driving	Predator Prey	Pursuit Evasion	DTC
Properties	Cooperative	x			x	x				x			x		x
	Mixed		x			x			x	x		x	x		
	Competitive			x			x	x			x			x	
	Symmetric roles	x	x	x	x	x			x	x		x	x		x
	Asymmetric roles						x	x			x			x	
	Discrete Actions	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Continuous Actions											x	x	x	x
	Discrete Observations	x	x	x	x	x	x	x	x	x	x				
	Continuous Observations											x	x	x	x
	Pixel Observations					x	x	x	x	x	x				
Concepts	Temporal Coordination	x			x	x			x	x		x	x		x
	Spacial Coordination				x	x			x	x		x	x		x
	Reciprocity					x									
	Fair Resource Sharing					x				x		x	x		
	Deception				x		x	x			x			x	
	Convention following								x			x			
	Nested-Reasoning		x	x			x	x			x			x	

of heuristic and deep RL policies. In this section we provide some details on the general training procedures used for the deep RL policies. We also go into greater detail about the policy populations used in our experiments. Over time we expect to update the set of policies included in POSGGym, for the full up-to-date list please check out the documentation at https://posggym.readthedocs.io/. All code used for training the RL policies is available at https://github.com/RDLLab/posggym-baselines.

C.1 Reinforcement Learning Policy Training

Every RL policy included with POSGGym to-date uses a LSTM actor-critic neural network architecture and was trained using Proximal Policy Optimization (PPO) (Schulman et al. 2017). LSTM's allow each policy to be conditioned on histories of action and observations which generally work more effectively in partially observable environments. The specific architecture used consisted of a fully-connected network (FCN) trunk, followed by a single layer LSTM, and then separate FCN actor and critic heads. The specific neural network architecture and training hyperparameters for each environment are shown in Table 5 and Table 6. We used commonly used hyperparameters values for the grid-world problems, since we found these generally worked. For the continuous environments, some hyperparameter tuning was conducted to select appropriate values. The number of training steps was chosen such that policies could train until convergence, as indicated by their learning curves.

We used different multi-agent training schemes depending on the environment, while the same RL algorithm and neural network architecture was used for each individual policy. The two schemes we used were K-Level Reasoning (KLR) (Cui et al. 2021) and independent self-play with best response (SP-BR, commonly referred to as Independent PPO when using PPO as the RL algorithm) (de Witt et al. 2020). We used these two methods as they have been extensively studied (Lanctot et al. 2017; Cui et al. 2021; Tesauro 1994; de Witt et al. 2020), are simple to implement (and thus replicate), and produced diverse populations of policies when combined with policy pruning to remove similar policies. Figure 9 provides a visualization of the training schema used. The following sections contain a high level overview of each scheme.

K-Level Reasoning In K-Level Reasoning (KLR) training policies are trained in a hierarchy, the level $K = 0$ policy is trained against a uniform random policy, level $K = 1$ is trained against the level $K = 0$, and so on with the level K policy trained as a best response to the level $K - 1$ policy for $K > 0$. Finally, the best-response policy K_{BR} is trained against all K level policies, excluding the random policy and the K_{BR} policy itself. In our implementation we used the Synchronous KLR Best-Response (SyKLRBR) training method (Cui et al. 2021) which trains all policies synchronously and was shown to converge in less total wall time and lead to generally more robust policies.

Self-Play Self-play training involves training independent policies against themselves (Tesauro 1994; de Witt et al. 2020). For asymmetric environments this meant training a

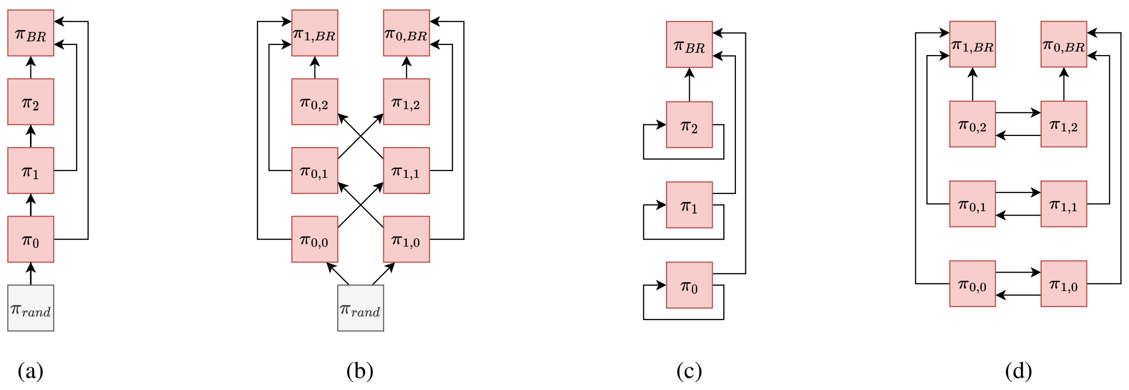


Figure 9: Multi-agent training schemas used for generating RL policies for POSGGym environments. (a) KLR in symmetric environment, (b) KLR in asymmetric environment with two agents, (c) self-play in symmetric environment, (d) self-play in asymmetric environment with two agents. Each box is an independent policy and arrows indicate which policy a given policy was trained against. Figure adapted from (Cui et al. 2021).

Table 5: Training hyperparameters for POSGGym Agents RL policies in grid-world environments.

Hyperparameter	Driving	LBF & Predator Prey	Pursuit Evasion
Training steps	32M	100M	10M
Trunk layer sizes	[64, 64]	[64, 64]	[64, 32]
LSTM size	64	64	256
Head layer sizes	[64]	[64]	-
γ	0.99	0.99	0.99
Learning rate	0.0003	0.0003	0.0003
GAE λ	0.95	0.95	0.95
Batch size	6144	6144	2048
Mini-batch size	2048	2048	256
Rollout length	64	64	100
Update epochs	2	2	2
BPTT seq. length	10	10	20
Entropy bonus	0.01	0.01	0.001
Value function coeff.	0.5	0.5	1.0
Clip parameter	0.2	0.2	0.3
Global gradient clip.	10	10	10

set of policies one for each agent in the environment for each training seed. While for symmetric environments a single policy is used by all agents in the environment. In self-play Best-Response (SP-BR) an additional best-response policy π_{BR} is trained against a uniform distribution over all the independent policies trained.

Table 6: Training hyperparameters for POSGGym Agents RL policies in continuous environments.

Hyperparameter	Value
Training steps	100M
Trunk layer sizes	[256, 256]
LSTM size	256
Head layer sizes	-
γ	0.99
Learning rate	0.0003
GAE λ	0.95
Batch size	65,536
Mini-batch size	2048
Rollout horizon	100
Update epochs	2
BPTT seq. length	20
Entropy bonus	0.001
Value function coeff.	1.0
Clip parameter	0.5
Global gradient clipping	10

D Experiment Environments and Populations

Figure 10 shows the five environments used in our experiments.

For our experiments we used a set P of 10 to 12 policies for each environment we tested in.

CooperativeReaching P consisted of 11 heuristic policies `H[1-11]`. With $P_0 = \{$`H1, H2, H3, H4, H5`$\}$ and $P_1 = \{$`H6, H7, H8, H9, H10, H11`$\}$. These policies were based on prior work (Rahman et al. 2023) with some adjustments made to ensure the population had diverse returns Figure 11a.

Driving P consisted of 10 policies: five heuristic and five RL trained policies. $P_0 = \{$`A0, A40, A60, A80, A100`$\}$ was made up of the heuristic policies, while $P_1 =$

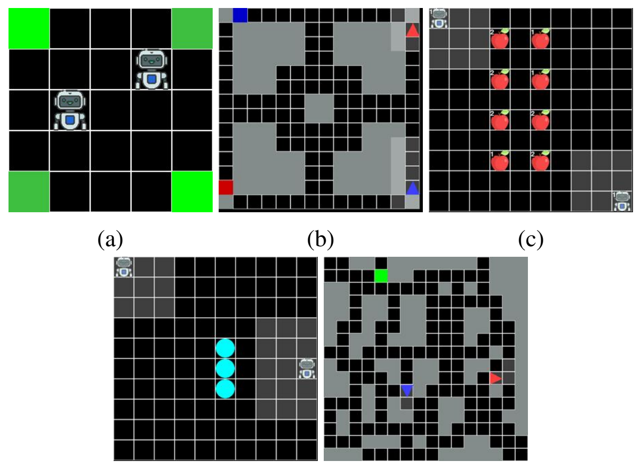

(a) (b) (c)
(d) (e)

Figure 10: Our experiments used a diverse set of environments, including cooperative ((a) *CooperativeReaching* and (d) *PredatorPrey*), mixed ((b) *Driving* and (c) *LevelBasedForaging*), and competitive ((e) *PursuitEvasion*) scenarios. Since it is asymmetric we use two versions of *PursuitEvasion*: *i0* and *i1* where the planner controls the pursuer (blue) and evader (red), respectively

$\{$RL1, RL2, RL3, RL4, RL5$\}$ contained all the RL policies. Each heuristic policy followed the shortest path from the agent's start position to the goal but differed on how aggressive they were, from least aggressive A0 to most aggressive A100. The aggressiveness of a policy controlled how far away another agent had to be within the agent's field of vision before the policy would stop the agent's vehicle from moving. A0 would stop if another agent was observed anywhere and would only continue once that agent was out of view. Conversely, A100 would continue along the shortest path irrespective of how close another observed agent was. A[40-80] followed policies between the two extremes. The RL policies RL[1-5] were produced by first training six policies using SP-BR and then pruning away any similar policies based on pairwise returns to give the final set of five policies. The pairwise returns for each policy in the population P for this environment are shown in Figure 11b.

LevelBasedForaging P consisted of 10 policies: five heuristic and five RL trained policies. $P_0 = \{$H1, H2, H3, H4, H5$\}$ contained the heuristic policies, while $P_1 = \{$RL1, RL2, RL3, RL4, RL5$\}$ contained all the RL policies. The heuristic policies were based on prior work (Rahman et al. 2023), adapted to deal with partial observability. We pruned many of the heuristic policies used in the prior work as we found that they resulted in similar behaviours based on their returns. The five heuristic policies used were:

- H1 always goes to the closest observed food, irrespective of the foods level.
- H2 goes towards the visible food closest to the centre of visible players, irrespective of food level.
- H3 goes towards the closest visible food with a compatible level.
- H4 selects and goes towards the visible food that is furthest from the center of visible players and that is compatible with the agents level.
- H5 targets a random visible food whose level is compatible with all visible agents.

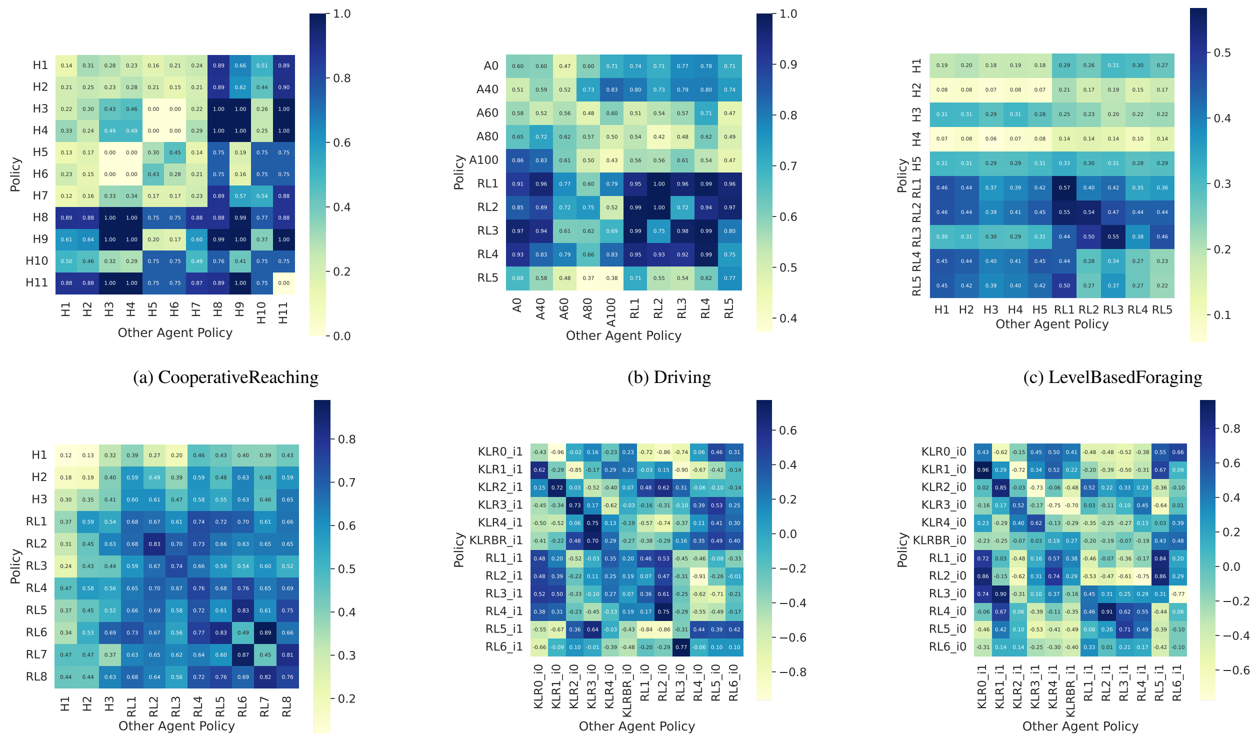
For the RL policies we trained a population of 13 RL policies including six using SB-BR and seven using SyKLRBR (up to $K = 5$). The resulting five RL policies RL[1-5] were found by pruning away similar policies from the full set of 13 policies. To do this the policies were clustered based on their pairwise returns then a single policy from each cluster was chosen. The pairwise returns for each policy in the population P for this environment are shown in Figure 11c.

PredatorPrey P consisted of 11 policies: three heuristic and eight RL trained policies. $P_0 = \{$H1, H2, H3, RL1, RL2$\}$ was made up of a mix of heuristic and RL policies, while $P_1 = \{$RL3, RL4, RL5, RL6, RL7, RL8$\}$ contained the remaining RL policies. The heuristic policies were chosen based on trying various heuristics and selecting those that had diverse pairwise returns. The three heuristic policies used were:

- H1 moves towards closest observed prey, closest observed predator, or explores randomly, in that order.
- H2 moves towards closest observed prey, closest observed predator, or explores in a clockwise spiral around arena, in that order.
- H3 moves towards closest observed prey to the closest observed predator or explores in a clockwise spiral around arena, in that order.

For the RL policies we followed an identical protocol to the LevelBasedForaging environment; first training 13 policies using SP-BR and SyKLRBR and then pruning similar policies to produce the final population of RL policies. The pairwise returns for each policy in the population P for this environment are shown in Figure 11d.

PursuitEvasion (evader "0" and pursuer "1") For both agents $X \in 0, 1$, P consisted of 12 RL policies. $P_0 = \{$KLR0_iX, KLR1_iX, KLR2_iX, KLR3_iX, KLR4_iX, KLRBR_i$X\}$ was a population of KLR policies trained using SyKLRBR, while $P_1 = \{$RL1_iX, RL2_iX, RL3_iX, RL4_iX, RL5_iX, RL6_i$X\}$ contained RL policies trained using a mix of SP-BR and SyKLRBR. For the RL policies a population of 30 SyKLRBR policies (five separate populations of six policies with up to $K = 4$) and five self-play (no best-response) policies were trained. The most diverse (based on pairwise returns) SyKLRBR population of six policies was then selected for P_0. P_1 was then chosen by pruning away similar policies from the remaining 29 SyKLRBR and self-play policies, based on pairwise returns. The pairwise returns for each policy in the population P for this environment are shown in Figure 11e and Figure 11f.



(a) CooperativeReaching (b) Driving (c) LevelBasedForaging

(d) PredatorPrey (e) PursuitEvasion i0 (Evader) (f) PursuitEvasion i1 (Pursuer)

Figure 11: Payoff tables for POSGGym agent policies for environments used in the experiments. Each table shows the mean returns for the row policy when paired with the column policy after 1000 episodes.

E Planning Experiment Details

The implementation of the planning methods used in our experiments were based the implementations used in prior work (Schwartz, Kurniawati, and Hutter 2023). The hyperparameters used by each method are shown in Table 7 and for all methods we used normalized Q-values during planning as per (Schrittwieser et al. 2020). For methods that used UCB (INTMCP, IPOMCP, POMCP) we used rollouts using a random policy for leaf node evaluations. For INTMCP and POMCP actions for the other agent during rollouts were chosen using a random policy, while for IPOMCP they were chosen using the other agent policy sampled from the root belief. POTMMCP used the value function from its meta-search policy for leaf node evaluation where available, otherwise used rollouts using the meta-policy for action selection. We used rejection sampling for all methods for belief reinvigoration after each update. This was used over other methods such as weighted particle filtering (Sunberg and Kochenderfer 2018) as it did not require access to an observation function. All code is available at https://github.com/RDLLab/posggym-baselines

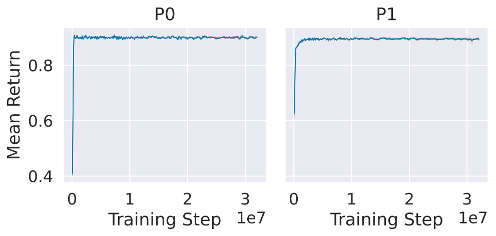
Table 7: Hyperparameters for different planning methods used in our experiments. S is the search time used, for our experiments we used $S \in [0.1, 1, 5, 10, 20]$ s.

Hyper-Parameter	Value
Discount (γ)	0.99
Discount horizon (ϵ)	0.01
Belief particles	$\lceil 100 \times S \times 1.16 \rceil$
C_{PUCB}	1.25
PUCB exploration (λ)	0.25
C_{UCB}	$\sqrt{2}$

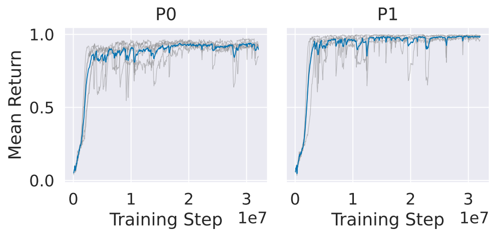
F Learning Experiment Details

For the learning based method (RL-BR) used in our experiments we trained a single deep RL policy $\pi_{BR,k}$ as a BR against each population $P_k \in [P_0, P_1]$ of other agents in each environment. PPO (Schulman et al. 2017) was used as the RL algorithm. During training at the start of each episode a policy for the other agent π_{-i} was sampled from a uniform distribution over the population being trained against and this policy was used to select actions for the other agent $-i$ while actions for the ego agent i were sampled the BR policy $\pi_{BR,k}$. In this way each policy $\pi_{BR,k}$ was trained to maximize its expected return against the uniform mixture over the population P_k.

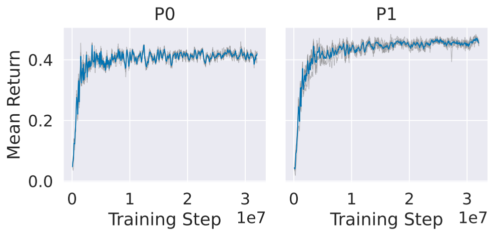
The BR policy used the same neural network architecture used by the population policies (see Section C.1) with a FCN trunk, followed by an LSTM layer, then finally separate FCN policy and value function heads. The hyperparameters are shown in Figure 8. We trained five separate policies us-



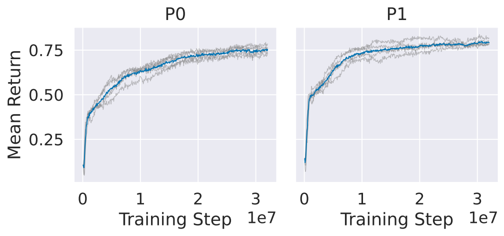
(a) CooperativeReaching-v0



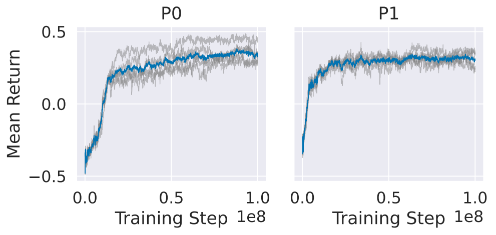
(b) Driving-v1



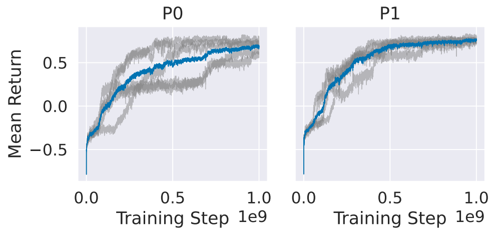
(c) LevelBasedForaging-v3



(d) PredatorPrey-v0



(e) PursuitEvasion-v1 i0 (Pursuer)



(f) PursuitEvasion-v1 i1 (Evader)

Figure 12: Learning curves for the BR RL policy in each environment against each population of other agent policies P_0, P_1. Grey lines show the mean episode return throughout training for each of the five different seeds. The blue line shows the average across seeds.

ing different seeds for each combination of population and environment with each policy being trained for up to 100M steps. Figure 12 shows the learning curve for each policy as well as the average learning curve across seeds for each population and environment.

G Generalization Results

Figures 13 and 14 shows the In- vs Out-of-distribution results for each method used in our experiments.

H Belief Accuracy Results

Figures 15 and 16 show the belief accuracy for the combined method throughout an episode in each environment.

Note, for the PursuitEvasion environment in 15 we see belief state accuracy decreases over time for both in- and out-of-distribution settings. This is expected and is due to naturally growing uncertainty over time inherent to the environment. In particular, in PursuitEvasion both agents start off knowing each others initial location but then receive no or very indirect observations of each other until either the pursuer perceives the evader or the evader reaches the goal,

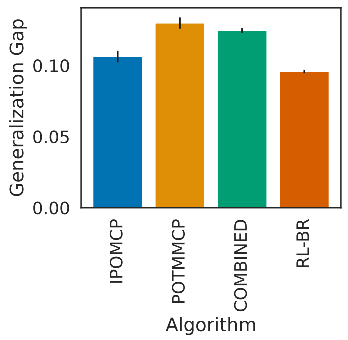


Figure 13: Gap between in- and out-of-distribution mean normalized returns of each method averaged across all environments. For planning and combined methods results using the maximum search time (20 s) are shown.

Table 8: Training hyperparameters for RL-BR policies used in our experiments.

Hyperparameter	Value
Training steps	100M (PursuitEvasion-v1 i0) 1B (PursuitEvasion-v1 i1) 32M (all other environments)
Parallel workers	32
Trunk layer sizes	[64, 64]
LSTM size	64
Head layer sizes	[64]
Discount (γ)	0.99
Learning rate	3×10^-4
GAE λ	0.95
Batch size	65536
Mini-batch size	2048
Rollout horizon	64
Update epochs	2
BPTT sequence length	10
Entropy bonus	0.01
Value function coeff.	0.5
Clip parameter	0.2
Global gradient clipping	10

at which point the episode ends. This growing uncertainty is in contrast to all other environments, including LevelBased-Foraging, where we expect more certainty over time since agents observe each other more directly. Additionally, both planning and RL methods are affected the same by the growing uncertainty in the PursuitEvasion environment as it is independent of the algorithm. This compares with the poor belief accuracy in LevelBasedForaging which is due to poor belief approximation stemming from the planning methods belief representation.

I Experiment Episode Lengths

Figure 17 shows the average episode lengths of each method in each environment. Maximum episode length is 50 for all environments except PursuitEvasion-v1, where it is 100. PursuitEvasion-v1_i1 had the longest effective planning horizon. All other environments had shorter effective planning horizons due to the availability of more frequent positive rewards for the agent.

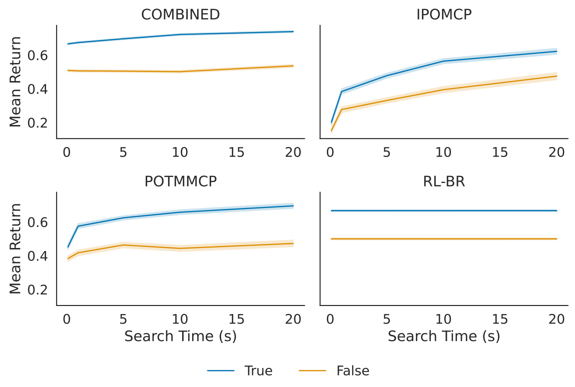


Figure 14: Average in- (True) versus out-of-distribution (False) performance for each method based on search time.

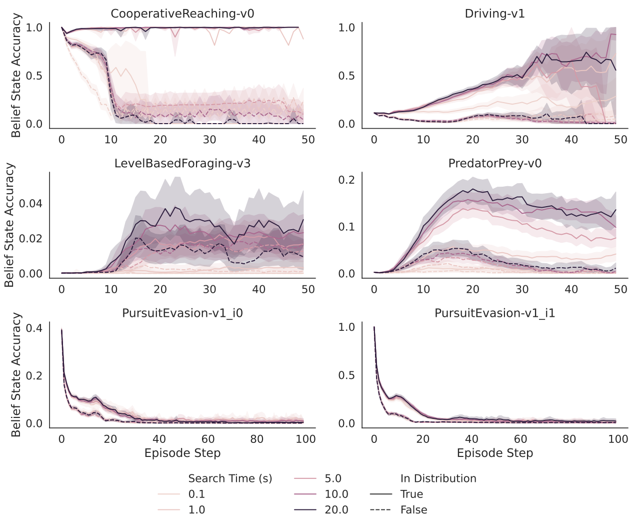


Figure 15: Probability assigned by the combined method's belief to the true environment state during an episode. In general belief accuracy is significantly worse in the out-of-distribution setting, and also in the LevelBasedForaging-v3 environment where the initial belief size is largest (y-axis scales differ between plots).

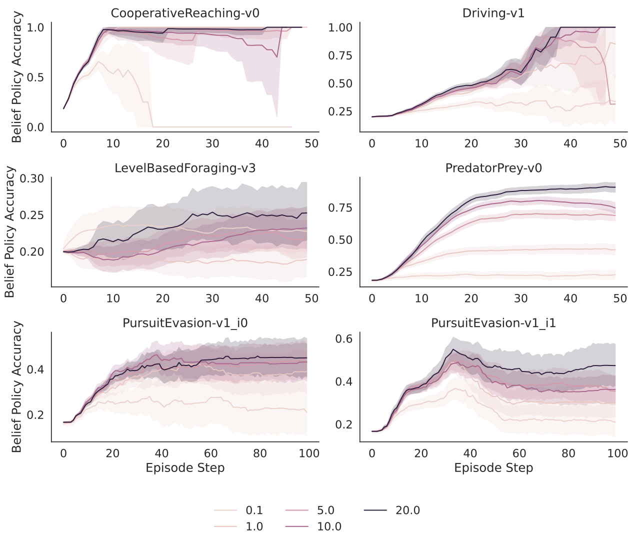


Figure 16: Probability assigned by the combined method's belief to the true policy of the other agent throughout an episode in the in-distribution setting. In general belief accuracy increases with search budget, and as the episode progresses and the agents have more interactions

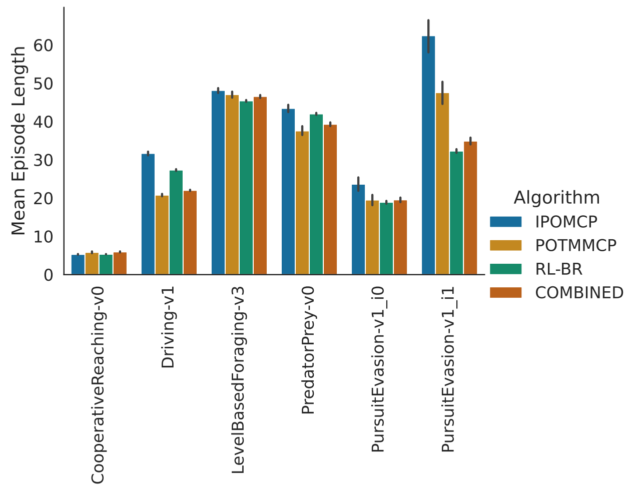


Figure 17: Mean episode length for each algorithm in each environment for the in-distribution setting. Results for planning and combined methods are using the maximum search time (20 s).