sIPOMDPLite-Net: Lightweight Self-Interested Learning and Planning in Partially Observable Multiagent Settings with Sparse Interactions

by

Gengyu Zhang

(Under the Direction of Prashant Doshi)

Abstract

This work introduces sIPOMDPLite-net, a deep neural network (DNN) architecture for agent control in partially observable multiagent settings with sparse interactions between agents. The network represents a new method for planning in contexts modeled by the interactive partially observable Markov decision process (I-POMDP) Lite framework with non-cooperative sparse interactions, which facilitates self-interested planning in settings shared with other agents more tractable than the well-known I-POMDP framework. The network uses fully-differentiable value iteration networks to simulate the solution of nested MDPs, which I-POMDP Lite attributes to the other agent to model its behavior, avoiding the need for involving non-differentiable techniques such as particle filtering to model the other agents more generally. We train sIPOMDPLite-net on a small two-agent tiger-grid problem, for which it accurately learns the underlying model and near-optimal policy, and the trained model continues to perform well on much larger and complex grids. As such, sIPOMDPLite-net shows good transfer capabilities and offers a lighter learning and planning approach for individual agents in multiagent settings.

Index words: Interactive POMDP Lite, Sparse interactions, Learning for planning

sIPOMDPLite-Net: Lightweight Self-Interested Learning and Planning in Partially Observable Multiagent Settings with Sparse Interactions

by

Gengyu Zhang

B.E., China University of Geoscience, China, 2016

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2021

sIPOMDPLite-Net: Lightweight Self-Interested Learning and Planning in Partially Observable Multiagent Settings with Sparse Interactions

by

Gengyu Zhang

Major Professor:	Prashant Doshi
Committee:	Frederick Maier Tianming Liu

Electronic Version Approved:

Ron Walcott
Dean of the Graduate School
The University of Georgia
December 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multiagent sequential decision-making under uncertainty is one of the most common but computationally challenging problems we face in the modern world. From maneuvering troops on the battlefield to our daily driving on roads with complex traffic, we regard them as multiagent sequential decision-making issues and manage to address them by pursuing their optimal solutions. Nevertheless, it is computationally hard to solve such problems, especially when they are under partial observability and participants of the multiagent systems are non-cooperative. Hence, designing self-interested agents that effectively and efficiently plan their actions under such formidable conditions has become an appealing cutting-edge research area.

Based on the underlying models they attribute to characterize agents' interactions and their algorithms to solve them, we can generally divide the existing approaches into game-theoretic and decision-theoretic frameworks. The game-theoretic frameworks represent the interactions in partially observable stochastic games (POSGs) and solve them by computing their Nash equilibria [1, 2]. In particular, they lay the foundation for majority of the well-known recent work empowering multiagent reinforcement learning (MARL) with deep learning (DL) [3, 4, 5]. However, the decision-theoretic frameworks extend single-agent decision-theoretic planning frameworks like MDP and POMDP to characterize interactions by explicitly including other agents' types in the

inference and solving the expanded model based on that. Arguably, the representative one for self-interested planning in non-cooperative settings is the I-POMDP [6, 7, 8, 9, 10]. Unfortunately, exactly solving I-POMDPs is possible but prohibitively expensive due to the exacerbated curses of dimensionality and history. The *curse of dimensionality* is due to the joint space of physical states and types of other agents that interactive beliefs need to account for, while the *curse of history* is rooted in the exponentially-growing policy space with the length of the planning horizon. To mitigate the framework's computational complexity from the ground up, Hoang and Low [11] introduced the more pragmatic IPOMDP-Lite framework, which significantly reduces the complexity of predicting the other agent's actions – a key complexity driver – by ascribing a nested MDP instead of POMDPs or I-POMDPs.

We also notice that in a large portion of real-world multiagent systems, agents tend to act alone most of the time and interact only when necessary, which is known as sparse interaction. The sparse interaction in multiagent systems has been a subject with a long history of research. Melo and Veloso [12, 13] tried to simplify the traditional decentralized MDPs (Dec-MDPs) under situations where the demand of coordination is limited to some specific subset of the state space. Based on this, they contributed the Dec-SIMDP, a decision-theoretic model for decentralized sparse-interaction multiagent systems that explicitly distinguishes the situations in which agents must coordinate from those in which they are safe to act independently. They also demonstrated an RL algorithm where agents learn both individual policies and when and how to coordinate. Yu, Zhang, and Ren [14, 15] proposed a learning approach for loosely-coupled multiagent systems that explicitly quantifies and dynamically adapts agent independence during learning so that agents learn to make trade-offs between single-agent learning and coordinated learning. Hu, Gao, and An [16] managed to utilize learned single-agent knowledge and to transfer it to multiagent settings via game-theoretic MARL. They proposed three transfer mechanisms, including directly transferring agents' single-agent value functions, only transferring value functions in states where the environmental dynamics change slightly, and transferring models by abstracting the one-shot game in each

state.

A recent vein of investigations has introduced deep DNN-based analogs of well-known planning frameworks, including value iteration networks (VIN) [17, 18] as the counterpart to MDPs, QMDP-net [19] as the counterpart to an approximation of POMDPs, and IPOMDP-net [20] as counterpart to the I-POMDP framework. The architectures learn an internal model of the decision-making problem and simulate the planning steps of the corresponding frameworks. Analogously to model-based reinforcement learning [21, 22, 23], these architectures bring the benefit of combined learning and planning directly on state-action-reward trajectories as input data. Furthermore, they have also demonstrated good transfer planning capabilities, with learned NNs generating high-quality plans for larger instances of the same domain, potentially lending themselves to meta-learning and subsequent few-shot learning and planning for multiple domains.

Continuing along this vein, we introduce sIPOMDPLite-net, a novel DNN architecture for learning and planning in stochastic settings shared with other agents as modeled by the I-POMDP Lite with sparse interactions. We construct the network architecture to represent a policy for a class of tasks, where we explicitly embed the planning model in the network architecture and train the network using imitation learning. First, we generate expert trajectories for a subset of tasks by solving their underlying I-POMDP Lite models and getting the optimal policy. Then, we sample trajectories of joint actions based on the policy and simulate them in the tasks, receiving corresponding trajectories of observations. Then, we train the network on the tasks with the expert demonstration, where we try to minimize the cross-entropy between the output action trajectories by our network and the expert trajectories.

As a vital reference of our work, the IPOMDP-net[1] [20] encodes the I-POMDP framework and solves it via I-PF [8]. It keeps a set of particles representing interactive beliefs throughout the reasoning and embeds a nested QMDP planner to predict other agents' actions. However, the network

[1]The source code of IPOMDP-net is unavailable, so we do not know its implementational details of the network architecture and experiments.

suffers from the *curse of nested reasoning* [11] due to interactive beliefs. Besides, the particle space for applying I-PF is vast, but the particle number is limited. There is a dilemma between the prediction accuracy of belief distributions and computational complexities. Additionally, it is intractable to implement random sampling in a NN that requires full differentiability.

The sIPOMDPLite-net accomplishes the same mission with a concise structure and low computational complexity by selecting the I-POMDP Lite as the underlying framework instead of I-POMDP. Hence, we address the *curse of reasoning* by substituting interactive beliefs with beliefs over physical states, by which we replace the bulky interactive belief update filter with fully differentiable network structures and thus enable the end-to-end training.

We originally proposed the Tiger-grid problems, of which the details will be presented in Section 4.1, as a representative application domain to train and evaluate the sIPOMDPLite-net, showing its performance on the model learning, policy learning, and transfer capability. The domain generalizes the well-known partially observable Tiger problem to a 2D grid shared by two agents, where there is a door in each cell of the grid, but only one door hides a pile of gold that the two agents try to capture. The domain complicates the original Grid-world by introducing extra decisions in each cell. Hence, it is more difficult to be solved by classic planning frameworks, which also imposes a greater challenge to our network for learning to solve it.

We choose the Tiger-grid as our application domain because it retains key features of non-cooperative sparse interaction and spatial locality. First, the two agents do not interact with each other except that either of them finds the door hiding the gold and opens it. Second, the agents can only move to adjacent cells or stay at the previous cells by taking any valid actions in a step. Such setup is good for the model trained by sIPOMDPLite-net on relatively small environments to transfer to larger ones since the dynamics are limited in local regions, and thus the transition model learned for such local regions applies to everywhere else in the environment.

In the rest of this chapter, we will clarify our objectives in this thesis by proposing the network architecture and articulate our contributions. Finally, we briefly introduce the structure of this

thesis.

1.1 Objectives

In this work, we attempt to solve the multiagent sequential planning tasks with the following characteristics:

1. The observability of agents is always partial regardless of the state, where an agent only receives local observations for its underlying physical state at each time step but no direct perceptions towards other agents.

2. The common (multiagent) states shared by the agents are equal to the Cartesian product of their private (single-agent) states, such that agents' transition and reward functions as to common states and joint actions can be factorized into those for private states and actions.

3. The interactions between agents are sparse. In other words, among all available state-action combinations of a multiagent planning task, only a relatively small subset of them lead to interactions, where the transitions and rewards must be determined by agents' common states and joint actions.

4. Agents are non-cooperative. Each agent fights for their own good instead of team rewards. Besides, unlike fully-cooperative scenarios with sparse interactions, agents not intending to interact can still be impacted by others who cause an interaction, which we call a passive interaction. Hence, the conditions of interactions are not only the intersection of each agent's interactive state-action pairs but their union.

1.2 Contributions

We contribute mainly in the following aspects:

1. We integrate the self-interested I-POMDP Lite framework with non-cooperative sparse-interactions. Instead of sharing a common observation in interaction states as in the Dec-SIMDP, agents must determine whether an interaction occurs by modeling others and predicting their behaviors.

2. It improves on the related IPOMDP-net in significant ways. First, the sIPOMDPLite-net offers a fully differentiable solution in comparison to IPOMDP-net, which utilizes the interactive particle filtering [8]; the resampling used in particle filters is known to be not differentiable [24]. In contrast to IPOMDP-net, which maintains beliefs over the physical states and models ascribed to the other agent, sIPOMDPLite-net uses beliefs over the state only, thereby obviating the need for a particle filter and instead simulates the Bayes filter exactly. Consequently, all of its computations are differentiable similar to the VIN and QMDP-net, allowing for end-to-end training. Second, the sIPOMDPLite-net presents a more compact and lucid architecture that builds on the QMDP-net at the top level, simulating the planning for the subject agent while predicting the actions of the other agent using an architecture similar to the VIN. Both the compactness and differentiability lead to learning and planning that is more efficient and transparent than the IPOMDP-net.

3. We train sIPOMDPLite-net with expert trajectory data on a set of 6×6 grids and evaluate the transfer planning capability of the trained model on unseen grids of 7×7, 8×8, 10×10, and 12×12. We compare the performance between the expert policy obtained by solving the underlying I-POMDP Lite model and the policy learned by the network in three aspects – the success rate, false open rate, and collision rate. The results show that the learned policy continues to plan well in unseen 6×6 grids, where it has the success rate of 0.851 versus the expert policy's 0.858, the false open rate of 0.235 versus the expert policy's 0.208, and the collision rate of 0.070 versus the expert policy's 0.059. When the agents are situated in larger and more complex grids, the learned policy still performs comparably with its expert

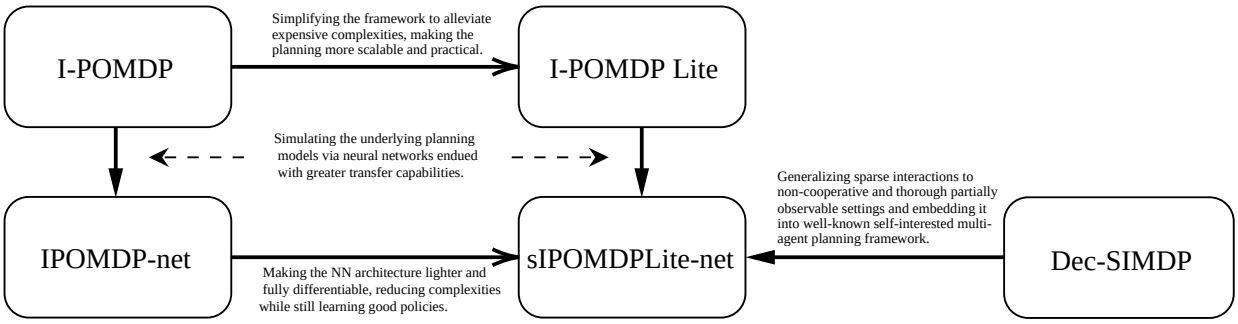


Figure 1.1: Relations between sIPOMDPLite-net and its important related work.

counterparts. What is most surprising is that in 10×10 and 12×12 grids, the success rates of the learned policy become even higher than those of the expert policy, where the results are 0.731 versus 0.712 and 0.743 versus 0.694. Furthermore, we evaluate the learned policy on much larger and realistically complex environments provided by a few LIDAR maps without further training the model on them. The learned policy performs comparably with the expert demonstrations. In two of the four environments, ACES and Orebro, the learned policy outperforms the expert policy in success rate with 0.842 and 0.950 versus 0.830 and 0.890. In the other two, though performing slightly worse, the learned policy still holds the matching success rate of 0.803 and 0.910 versus the expert's 0.820 and 0.940. The empirical result indicates that the network has successfully learned the essential elements of the problems, the other agent's strategy, and how to act near optimally in the problem.

1.3 Structure of the thesis

In this chapter, we have specified sIPOMDPLite-net's major application scenarios, the partially observable multiagent planning problems modeled by I-POMDP Lite with sparse interactions. Then, we reviewed related work known to resolve similar issues to ours, where we discussed their strengths and inadequacies in dealing with our task. Then, we introduced our work and listed the

key contributions. Next, we briefly preview the key points of the ensuing chapters.

Chapter 2 provides background on the I-POMDP Lite framework, the Dec-SIMDP framework, and the neural network architectures, VIN and QMDP-net. We build the sIPOMDPLite-net primarily based on factored I-POMDP Lite framework, where we refer to the model factorization based on a non-cooperative sparse interaction framework. Hence, reviewing the two frameworks helps provide the ground for our introduction of sIPOMDPLite-net in the next chapter. We then discuss the VIN and the QMDP-net that propose and finely exploit the learning-for-planning technique.

In *Chapter 3*, we first expound on the theoretical basis of sIPOMDPLite-net, deriving the factored modules of I-POMDP Lite. Then, we present the specific network architecture designed for approximating the framework, illustrating how each network component simulates the corresponding part of the underlying framework and fulfills the end-to-end training.

In *Chapter 4*, we propose two application domains, the Tiger-grid games and the real-life map navigation problems, to evaluate the sIPOMDPLite-net. We then elaborate on the details of training and experiment on the domains, followed by presenting the results. At last, we conduct the ablation studies to evaluate the critical roles that some sub-network structures play.

Chapter 5 concludes the thesis. We highlight our contributions while specifying the limitations of the work by far. Finally, we discuss possible directions of interest to improve our work in the future.

Chapter 2

Background

In this section, we demonstrate three fundamental backgrounds that lay the foundation for the sIPOMDPLite-net. First, I-POMDP Lite[11] is the underlying multiagent planning framework for which we build our NN analog. Second, the non-cooperative sparse interaction framework provides the theoretial basis for factorizing the framework models, which further helps define the sparse-interaction belief update function and Bellman equation. Finally, the representative learning-for-planning approaches inspire us with the network design, training data setup, and the learning scheme to employ.

2.1 I-POMDP Lite framework

Considering two self-interested agents, i and j, in a multiagent system, we define an I-POMDP Lite model with respect to i as a tuple $\langle b_0, S, A, \Omega_i, T_i, O_i, R_i, \hat{\pi}_j^L \rangle$. b_0 is the initial belief of agent i over common physical states S shared by two agents; Ω_i is agent i's local observation space; $A = A_i \times A_j$ is the set of joint actions, which can be represented as the Cartesian product each agent's individual action space; definitions of the transition function T, reward function R_i, and observation function O_i are similar to those of POMDPs, except that the transitions and rewards are

now for common states and joint actions instead of either agent's private ones. $\hat{\pi}_j^l$: $S \times A_j \to [0,1]$ represents the predictive probability of which j will select a_j in s.

The I-POMDP Lite proposes the nested MDPs to predict the other agent's intention while planning. We define a nested MDP for agent i reasoning at level l as $M_i^L = \langle S, A, T_i, R_i, \{\pi_j^l\}_{l=0}^{L-1} \rangle$, where S, A, T_i, and R_i refer to the same concepts as those in the top-level modeling module. $\{\pi_j^l\}_{l=0}^{L-1}$: $S \times A_j \to [0,1]$ is a recursive reasoning model of j at level $l < L$. The optimal $(k+1)$-step-to-go value function of M_i^l for agent i at level $l \geq 0$ satisfies the Bellman equation:

$$Q_i^{l,k+1}(s,a) = R_i(s,a) + \gamma \sum_{s' \in S} T_i(s,a,s') \max_{a_i' \in A_i} \sum_{a_j' \in A_j} \hat{\pi}_j^l(s', a_j') Q_i^{l,k}(s', a') \tag{2.1}$$

When $l = 0$, agent j's mixed strategy $\hat{\pi}_j^0$ is simply $|A_j|^{-1}$, implying that a_j is selected randomly in each state; otherwise, when $l = L$, where $L > 0$, there is $\hat{\pi}_j^L(s, a_j) = \sum_{l=0}^{L-1} \Pr(l) \pi_j^l(s, a_j)$, where $\Pr(l)$ indicates the probability that agent j reasons at level l. By solving the nested MDP, we obtain j's optimal actions $Opt_j^l(s)$, based on which we define the corresponding reasoning model as

$$\pi_j^l(s, a_j) = \begin{cases} |Opt_j^l(s)|^{-1} & a_j \in Opt_j^l(s), \\ 0 & \text{otherwise} \end{cases}$$

Provided with the full observability of nested MDPs, agent i need not model agent j's belief as in I-POMDP. Hence its own belief reduces to a probabilistic distribution over $\Delta(S \times A_j)$. We can further factorize this to separate out the belief over physical states only, $b_i(s, a_j) = b_i(s) \hat{\pi}_j^L(s, a_j)$. As such, we essentially reduce the problem to POMDP planning embedded with the other agent's mixed strategy. The belief update function is:

$$b_i'(s') = \eta O_i(s', a_i, o_i') \sum_{s, a_j} T_i(s, a, s') \Pr(a_j | s) b_i(s) \tag{2.2}$$

The optimal value function in Bellman equation is:

$$Q_i(b_i, a_i) = \sum_{s,a_j} b_i(s) \left\{ R_i(s,a) Pr(a_j|s) + \gamma \sum_{o_i} \Pr(a_j, o_i|s, a_i) \max_{a_i} Q_i \Big[SE(b_i, a, o_i), a_i \Big] \right\} \tag{2.3}$$

2.2 Non-cooperative sparse interaction framework

In a multiagent system where agents interact sparsely, they do not depend on each other in states without interaction, so their reasoning is equivalent to single-agent POMDP. Thus, it might consume considerable unnecessary computing space if we continue to adopt the general I-POMDP Lite framework for such states, considering the size gap between common and individual state space and that between joint and individual action space. Instead, we can split out the subset of states where interaction might happen and apply the general framework. Then, it is safe for agents to stick to their single-agent policy in the rest states.

We formally define a set of boolean functions to indicate if a given state-action pair $(s, a) \in P_R^I$, and $(s, a) \in P_T^I$, respectively:

$$\mathcal{X}_{P_R^I}(s,a) = \begin{cases} 1 & \text{if } (s,a) \in P_R^I \\ 0 & \text{otherwise} \end{cases}, \quad \mathcal{X}_{P_T^I}(s,a) = \begin{cases} 1 & \text{if } (s,a) \in P_T^I \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Generally, indicator functions for the transition and the reward function are distinct, but they can be identical in practice, i.e., $P_R^I = P_T^I$.

Provided with the indicator functions, we represent the factorization of the agent i's reward function R_i, transition function T_i, and agent j's reward function in the original I-POMDP Lite as

follows:

$$\begin{aligned} R_i(s,a) &= \left[1 - \mathcal{X}_{P_R^I}(s,a)\right]\overline{R_i}(s_i,a_i) + \mathcal{X}_{P_R^I}(s,a)R_i(s,a) \\ R_j(s,a) &= \left[1 - \mathcal{X}_{P_R^I}(s,a)\right]\overline{R_j}(s_j,a_j) + \mathcal{X}_{P_R^I}(s,a)R_j(s,a) \end{aligned} \tag{2.5}$$

$$T_i(s,a,s') = \left[1 - \mathcal{X}_{P_T^I}(s,a)\right]\overline{T_i}(s_i,a_i,s_i')\overline{T_j}(s_j,a_j,s_j') + \mathcal{X}_{P_T^I}(s,a)T_i(s,a,s') \tag{2.6}$$

Where $\overline{R_i}$, $\overline{R_j}$, $\overline{T_i}$, and $\overline{T_j}$ are single-agent models that equal to their multiagent counterparts in non-interaction situations.

2.3 Learning for planning

Learning for planning is a school of methods applying deep learning to sequential planning problems. Unlike common deep reinforcement learning approaches that learn values or policies for planning optimal actions by integrating information over the whole past states (observations if it is under partial observability) and actions, the learning for planning formulates problems referring to existing decision-theoretic planning frameworks.

When applying the learning for planning methods, we explicitly embed underlying models of such planning frameworks, such as the Markovian transition model and the additive or discounted rewards, to the DNN. We select network structures performing consistent operations with them to guarantee that the models are learned to approximate their underlying counterparts well. Furthermore, the whole network is constructed to represent the policy for the problems. Considering the recursive property in the planning solution or the belief update under partial observability, we typically connect exact planning algorithms with RNNs. For instance, when dealing with value iterations, we can imitate the algorithm with an RNN, where we set the state utilities for each iteration as a hidden state of the RNN. Similarly, when it comes to the belief update, we can regard

the belief filtered for each time step as a hidden state.

Let us take the QMDP-net [19] as an instance. The authors connect a parameterized POMDP model with the QMDP algorithm that solves the model and embeds both in a fully-differentiable recursive policy network. Furthermore, the target planning problems they use to train and evaluate the NN are generalized and represented as a set of parameterized tasks, $\boldsymbol{\Theta}$, which restricts the policy class over the problems and thus improves the computational efficiency regarding the policy search. The QMDP-net encodes a POMDP model conditioned on $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, where:

$$M(\boldsymbol{\theta}) = (S, A, \Omega, T(\cdot|\boldsymbol{\theta}), O(\cdot|\boldsymbol{\theta}), R(\cdot|\boldsymbol{\theta}))$$

S, A, and Ω are the shared state space, action space, and observation space throughout the tasks; $T(\cdot|\boldsymbol{\theta})$, $R(\cdot|\boldsymbol{\theta})$, and $O(\cdot|\boldsymbol{\theta})$ denote the encoded transition function, reward function, and observation functions to learn from data. Specifically, as the network represents the QMDP policy, we can learn by minimizing the cross-entropy between a trajectory of time-consecutive actions output from the network and the trajectory of optimal actions demonstrated by an expert. As such, we learn the optimal policy as well as the underlying models.

The approach benefits the planning with better data efficiency than the model-free methods. By maintaining internal models, they exploit the underlying sequential nature of the decision-making, which approaches faster to the optimal policy and requires less training data.

The learning for planning is similar to the model-based RL in the above aspects. However, the latter always requires system identification to map observations to a transition model, which is then solved for a policy. Since accurate system identification is difficult in most realistic applications, model-free approaches are often preferred. Therefore, in awareness of the fact, the learning for planning methods combine model-based planning with model-free learning, taking advantage of both while averting the drawbacks.

The approximations of planning frameworks usually have strong transfer capabilities. After

training, the policy learns to map an observation to a hidden state inference and a planning computation relevant to the task and predicts action based on the resulting policy, which generalizes to unseen tasks.

Chapter 3

sIPOMDPLite-Net

In this chapter, we first expound on the theoretical basis of the sIPOMDPLite-net, showing the procedures of factorizing the I-POMDP Lite in its belief update and solution provided with sparse-interaction assumption. Then, we provide the details in designing the network's architecture based on the theoretical basis.

3.1 I-POMDP Lite with sparse interactions

We have shown the general factorization of the transition function (Eq. 2.6) and the reward function (Eq. 2.5). This section demonstrates their role in the belief update and the solution, where we substitute the original multiagent models with the factorized terms. We also try to interpret the process from a macroscopic perspective. We handle the belief update and the planning regarding the entire state and action space rather than a single element. Before all this, we first focus on the indicator functions critical to the sparse-interaction setting, elaborating on the relations and transformations between single-agent and multiagent indicators for transitions and rewards.

3.1.1 Indicator functions

We define the single-agent indicator functions based on the concept of active interaction. In non-cooperative multiagent systems, an agent's single-agent indicator function indicates the agent's private state-action pairs that lead to interactions that impact others. In this case, we do not include passive interactions, where the agent does not intend to interact with others but gets involved in interactions caused by others. Taking agent i and agent j as an example, we define their indicators in Eq. 3.1 and Eq. 3.2.

$$\mathcal{X}_{P_R^{I_i}}(s_i, a_i) = \begin{cases} 1 & \text{if } (s_i, a_i) \in P_R^{I_i} \\ 0 & \text{otherwise} \end{cases}, \quad \mathcal{X}_{P_R^{I_j}}(s_j, a_j) = \begin{cases} 1 & \text{if } (s_j, a_j) \in P_R^{I_j} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$$\mathcal{X}_{P_T^{I_i}}(s_i, a_i) = \begin{cases} 1 & \text{if } (s_i, a_i) \in P_T^{I_i} \\ 0 & \text{otherwise} \end{cases}, \quad \mathcal{X}_{P_T^{I_j}}(s_j, a_j) = \begin{cases} 1 & \text{if } (s_j, a_j) \in P_T^{I_j} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

While agents may have different single-agent indicators depending on their respective interaction triggering conditions, they share the same multiagent indicator by integrating the single-agent ones. We formulate such transformation as follows:

$$\begin{aligned} \mathcal{X}_{P_R^I}(s, a) &= \mathcal{X}_{P_R^{I_i}}(s_i, a_i) + \mathcal{X}_{P_R^{I_j}}(s_j, a_j) - \mathcal{X}_{P_R^{I_i}}(s_i, a_i)\mathcal{X}_{P_R^{I_j}}(s_j, a_j) \\ \mathcal{X}_{P_T^I}(s, a) &= \mathcal{X}_{P_T^{I_i}}(s_i, a_i) + \mathcal{X}_{P_T^{I_j}}(s_j, a_j) - \mathcal{X}_{P_T^{I_i}}(s_i, a_i)\mathcal{X}_{P_T^{I_j}}(s_j, a_j) \end{aligned} \tag{3.3}$$

3.1.2 Belief update

Given the sparse-interaction factorization for the transition function in Eq. 2.6 and the indicator transformation we derive in Eq. 3.3, we rewrite the original I-POMDP Lite belief update (Eq. 2.2)

as:

$$b_i'(s') = \eta O_i(s', a_i, o_i') \Bigg\{ \sum_{s_i} \Big[1 - \mathcal{X}_{P_T^{I_i}}(s_i, a_i)\Big] \overline{T}_i(s_i, a_i, s_i') \sum_{s_j, a_j} \Big[1 - \mathcal{X}_{P_T^{I_j}}(s_j, a_j)\Big] \overline{T}_j(s_j, a_j, s_j') \times \Pr(a_j|s) b_i(s) + \sum_{s, a_j} \mathcal{X}_{P_T^I}(s, a) T_i(s, a, s') \Pr(a_j|s) b_i(s) \Bigg\} \tag{3.4}$$

The first term of the sum enclosed in the curly brace corresponds to where no interaction occurs. In other words, the agents do not meet the condition such that $(s, a) \in P_T^I$. Thus, we can safely handle the belief update with each agent's private transition function. Here in Eq. 3.4, we introduce a two-step belief update for the non-interaction part, which first updates $b_i(s)$ with a_j and then updates the result with a_i. Specifically, we first multiply the belief with $1 - \mathcal{X}_{P_T^{I_i}}$, screening out the set of belief probabilities for situations where agent j does not interact with and affect agent i, i.e., $P_T^{I_j\complement}$. We then update it with $\overline{T}_j$. Similarly, in the second step, we filter the updated belief with $1 - \mathcal{X}_{P_T^{I_j}}$ before further updating the result with $\overline{T}_i$. As such, we complete the non-interaction belief update. As for the belief update for interactions, we first screen out the set of belief probabilities corresponding to P_T^I and update it with the multiagent transition function T_i directly.

We combine the result for both situations and finish the belief propagation with actions. The next step, belief propagation with observation, is consistent with the original I-POMDP Lite framework.

3.1.3 Solution

We select the multiagent QMDP algorithm [25, 26] to solve our models. As we know, QMDP solves POMDP approximately. It employs the basic MDP value iteration without including beliefs in its Bellman equation. Instead, the belief only shows up in the last step, which is used to weight

the computed Q values for the policy. Thus, while losing slight optimality, it provides a faster solution. On the other hand, I-POMDP Lite also reduces to a POMDP, except it is aware of other agents' policies. As such, we can retrofit the Bellman equation of QMDP slightly by introducing others' policies, which is, in fact, equivalent to the nested MDP's Bellman equation as shown in Eq. 2.1. Furthermore, constructing NN architecture for QMDP is much simpler than the original I-POMDP Lite planning, which also benefits the training.

Similar to what we have done to the belief update, we substitute the general factorization for R_i (Eq. 2.5) and T_i (Eq. 2.6) into the Bellman equation of QMDP and get:

$$Q_i^{k+1}(s,a) = \Big[1 - \mathcal{X}_{P_R^I}(s,a)\Big]\overline{R_i}(s_i,a_i) + \mathcal{X}_{P_R^I}(s,a)R_i(s,a) + \gamma\Bigg\{\Big[1 - \mathcal{X}_{P_T^I}(s,a)\Big] \times \sum_{s_i'}\overline{T_i}(s_i,a_i,s_i')\sum_{s_j'}\overline{T_j}(s_j,a_j,s_j')U_i^k(s') + \mathcal{X}_{P_T^I}(s,a)\sum_{s'}T_i(s,a,s')U_i^k(s')\Bigg\} \tag{3.5}$$

For the immediate reward, we merely refer to Eq. 2.5, while for the long-term reward, we factorize the term for non-interactions into agent i's and j's individual value iteration. We still organize the non-interaction part as a two-step procedure. The major difference from its counterpart in the belief update is the order of multiplying indicators. Bear in mind that the state-action pairs that the indicator functions indicate are always for the current time step. In the belief update, to get the belief for the next state s', we sum the transition probabilities times the current belief over all possible current states s. Thus, we need to first determine the exact belief for each s before being summed up. In the Bellman equation, nevertheless, it is just the opposite, where we get values for the current state s by summing up predicted values for all possible next state s'. Hence, given $U_i^k(s')$, we first get the current Q values by the two-step transition multiplication and then screen out values corresponding to non-interactions by the indicators. As Eq. 3.5 shows, the indicator function is put outside the sum.

Now that we obtain the Q values by executing the QMDP value iteration, it is natural to get the

optimal policy:

$$q_i(a_i) = \sum_{s,a_j} Q_i^K(s,a)\hat{\pi}_j(s,a_j)b_i(s) \tag{3.6}$$

3.2 Network architecture

This section illustrates the design of the sIPOMDPLite-net's architecture, showing each module's pivotal role in encoding the underlying models of the I-POMDP Lite framework with sparse interactions into a NN analog. We begin with an overview demonstrating the general architecture, establishing major functional modules, and clarifying their connections. Following that, we stipulate the standard input and output of the network. In particular, we elaborate on task parameters that serve as an essential part of training data. As for the demonstrations of major modules, per the theoretical exposition, we begin with the indicator transformer, showing the transformations between agents' single-agent and multiagent indicator functions in the form of tensors. Then, we highlight the belief update module and the value iteration solver module, expounding on their underlying logic of simulating the two most critical parts of the original framework. Following this, we present the nested MDP module and the QMDP planning module that share the structure of the value iteration solver for similar solutions.

For ease of discussion, we consider only the elementary setup in the rest of the section. The multiagent problem we attempt to solve involves only two agents, i.e., the subjective agent i and the objective agent j. Agent i reasons at level 1 and models agent j as a nested MDP reasoning at level 0; only one interaction triggering state-action pair exists, and the two agents share it.

3.2.1 Overview

We divide the whole sIPOMDPLite-net architecture into the inference part and the training part. The former, which explicitly simulates the composition of the underlying framework, generally

comprises a Bayesian filter module accounting for the belief update regarding the changing states over time and a solution module in charge of searching for the optimal policy for agents. The solution module further consists of the nested MDP solution module and the QMDP solution module. The former provides the predicted policy of agent j by recursively employing a nested sub-module, the MDP value iteration solver, to map both agents' immediate rewards and the adversary's lower-level policy to its optimal policy. The latter receives the current belief state from the Bayesian filter, the predicted policy of agent j from the nested MDP solver, and employs the same MDP value iteration solver to obtain agent i's optimal policy. Finally, it outputs an action based on this policy. As for the training part of the network, we discuss it in detail in 4.5.

After listing the main modules of the network and the connections between them, we specify the inputs and outputs of the sIPOMDPLite-net. The inputs consist of two types of data. One is task-dependent but time-independent, which we name as task parameters. The other is time-dependent, which we call trajectories. A task parameter set, denoted by Θ, covers all combinations of possible values of a wide variety of task features that vary within some specific set of tasks. Each such combination of features, i.e., a unique task parameter, which we denote by $\boldsymbol{\theta}$, unambiguously determines a task out of the task set. The task features capture the complete prior knowledge agents possess for this set of tasks, including those about environments and those about themselves.

Generally, a partial observable multiagent planning task requires the common state space of all involved agents as one of the environment-related prior knowledge and the initial belief over this state space shared by all agents as one of the agent-related prior knowledge. As Fig. 3.1 illustrates, the input task parameter $\boldsymbol{\theta}$ comprises the navigation maps, goal maps, and the common initial belief for all agents. A particular component in our task parameter regarding the sparse-interaction setting is the interaction indicator functions for the given task. We deem them priors since learning them as hidden models through the neural network remains challenging at this time.

We assume that the features of a task parameter $\boldsymbol{\theta}$ must contain the information that hides clues of an agent's underlying models, i.e., its transition, reward, and observation function. For example,

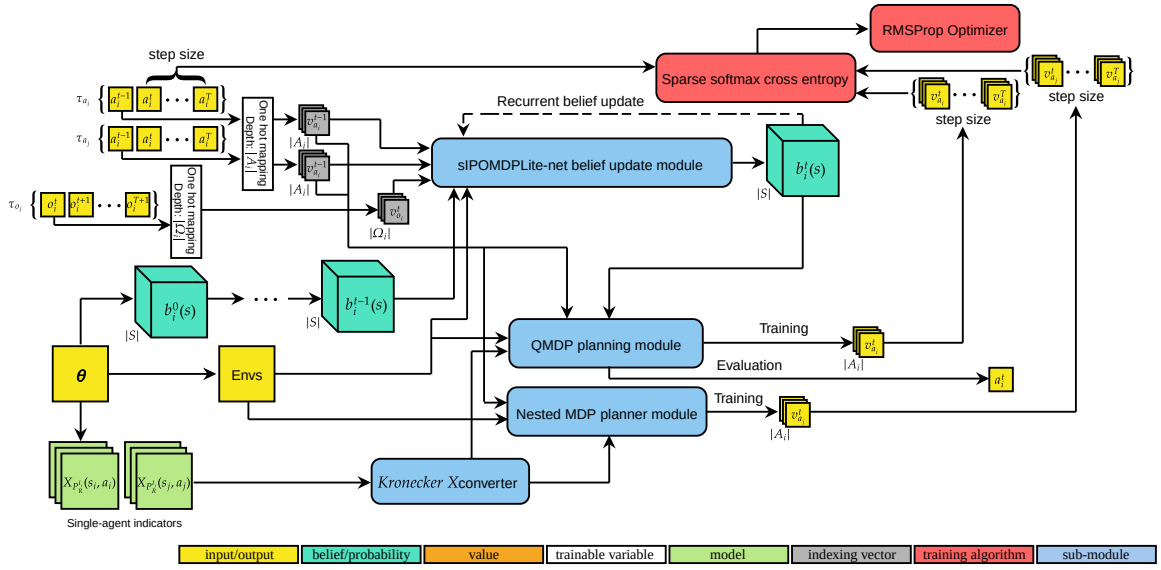


Figure 3.1: The top-level architecture.

in Tiger-grid problems, the grid map of an agent offers the basis for its observation function. On the other hand, the grid map and the goal map together determine its immediate rewards. Hence, when designing the network architecture for learning these underlying models, we explicitly condition them on specific components of $\boldsymbol{\theta}$.

When the task parameters uniquely define a task, our attention turns to the other input, i.e., trajectories. Similar to the related work about learning-for-planning approaches as mentioned in 2.3, sIPOMDPLite-net is essentially an RNN that recursively takes in trajectories of actions and observation and maps them to a policy while updating beliefs as hidden states. The input trajectories of our network consist of trajectories of both agents' actions and the subjective agent's local observations ordered by time step from the expert demonstration. The length of the trajectories depends on the step size of training the RNN provided as a hyperparameter.

Different from the I-POMDP Lite framework, where we treat the policy given by the nested MDP as a probability distribution over actions for each state, we provide sIPOMDPLite-net with the other agent's action directly, which helps the network learn the underlying models as accurate

as possible. From the perspective of training, which we specifically elaborate on in Section 4.5, explicit joint actions deliver clear information about the input to the network so that it learns which next action to take given a specific joint action. On the contrary, we cannot guarantee that the learned probabilities from the nested level of reasoning are correct in end-to-end learning. We then send such joint actions to the belief update, nested MDP, and QMDP planning module, training them separately to ensure models learned sufficiently accurately for approximating their underlying counterparts.

The output of sIPOMDPLite-net reflects the policy that it represents. The QMDP planning module outputs the subjective agent i's best action sampled from the policy in the training phase, while the nested MDP module outputs the objective agent j's optimal actions. Nonetheless, in the evaluation phase, the network only outputs agent i's action while transmitting agent j's action internally between the modules.

3.2.2 Indicator transformation module

As demonstrated in Eq. 3.4, we require both agents' single-agent indicators, i.e., $\mathcal{X}_{P_T^{I_i}}$ and $\mathcal{X}_{P_T^{I_j}}$, and the multiagent indicator they share, i.e., $\mathcal{X}_{P_T^I}$, in the belief update. As it is easier to transform $\mathcal{X}_{P_T^{I_i}}$ and $\mathcal{X}_{P_T^{I_j}}$ to $\mathcal{X}_{P_T^I}$ based on Eq. 3.3 but much harder to get the other way around, we include $\mathcal{X}_{P_T^{I_i}}$ and $\mathcal{X}_{P_T^{I_j}}$ in $\boldsymbol{\theta}$ and feed it to our network as priors. Then we get $\mathcal{X}_{P_T^I}$ based on Eq. A.1. The Kronecker indicator converter works as Fig. 3.2.

According to Eq. 3.5, however, we only require the multiagent reward interaction indicator, i.e., $\mathcal{X}_{P_R^I}$, instead of its single-agent counterparts, in the solution, so we directly input it to the network without any transformations.

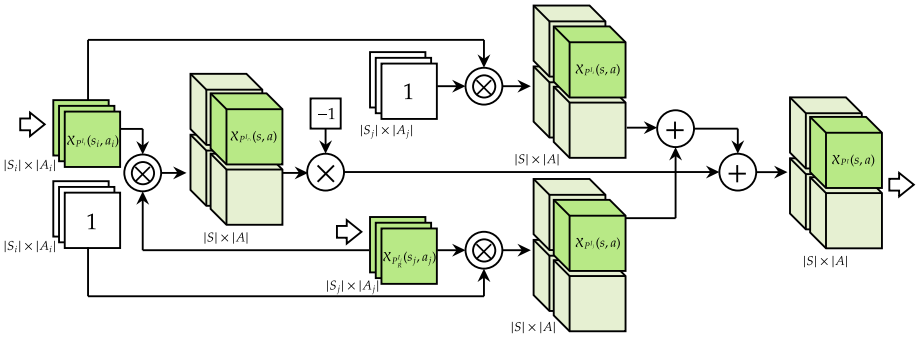


Figure 3.2: Three types of multiagent $\mathcal{X}$s.

3.2.3 Belief update module

Simulating the recursive update of beliefs, we construct the belief filter of sIPOMDPLite-net as a recurrent neural network (RNN), in which the belief plays the role of the hidden state. Fig. 3.3 illustrates a round of the recurrent belief update.

Our objective for this module is to accurately learn the underling models, including $T_i(s, a, s')$, $\overline{T}_i(s_i, a_i, s'_i)$, $\overline{T}_j(s_j, a_j, s'_j)$, and $O_i(s', a_i, o'_i)$. For some step of the recursion, the module maps a tuple of $\langle o'_i, a_i, a_j \rangle$ from expert trajectories, a belief tensor of agent i, $\boldsymbol{B}_S$, and $\boldsymbol{\theta}$ for the current task to another belief tensor, $\boldsymbol{B}'_S$.

Consistent with the framework, the first part of the belief update module works to update the input belief tensor $\boldsymbol{B}_S$ with the joint action for the step. Following Eq. 3.4, we first filter $\boldsymbol{B}_S$ for the belief over data points belonging to interactions with the transition interaction indicator, $\boldsymbol{X}_{P^I_T}$. Then, we update the resulting belief with the parameterized transition function of agent i, $T_i(s, a, s')$. Based on the principle of learning as accurate a model as possible, we enforce the NN analog to inherit properties of its underlying counterpart. Hence, we normalize the parameterized T_i over the dimension of s'. The network structure simulating the update with actions generally

implements the following operation:

$$\boldsymbol{B}'_{P_T^I} = \sum_{s \in S} \sum_{a_i \in A_i} \sum_{a_j \in A_j} \boldsymbol{B}_{P_T^I} T_i(s, a, s' | \boldsymbol{\theta}) \boldsymbol{v}_{a_i} \boldsymbol{v}_{a_j}$$

where $\boldsymbol{v}_{a_i}$ and $\boldsymbol{v}_{a_j}$ are indexing vectors for a_i and a_j.

As the equation shows, we update the belief for each $a \in A$. To obtain $\boldsymbol{B}'_{P_T^I}$ corresponding to the given $a = \langle a_i, a_j \rangle$, we multiply the output belief tensor with $\boldsymbol{v}_{a_i}$ and $\boldsymbol{v}_{a_j}$, respectively. The specific class of network structure with which $T_i(s, a, s' | \boldsymbol{\theta})$ is trained depends on the domain it deals with. Section 4.4.2 demonstrates an instance of solving Tiger-grid problems with spatial locality, where we encode the belief propagation into a convolutional layer with $T_i(s, a, s' | \boldsymbol{\theta})$ as the kernel.

On the other hand, we execute the two-step belief propagation for the non-interaction portion of $\boldsymbol{B}_S$. Based on Eq. 3.4, to update the belief with agent j's action in the first step, we first filter $\boldsymbol{B}_S$ for the belief probabilities over data points belonging to non-interaction parts with $1 - \boldsymbol{X}_{P_T^{I_j}}$, where $\boldsymbol{X}_{P_T^{I_j}}$ is j's single-agent transition interaction indicator. Then we update the resulting belief with the parameterized $\overline{T_j}$ and pick the $\boldsymbol{B}_S$ corresponding to the given a_j. In the second step, we repeat such a procedure with respect to agent i. Finally, we sum up the updated belief tensor for the interaction and non-interaction to complete the belief update with actions.

The next part is to update the resulting belief with the input observation for the step, o'_i. We condition the parameterized observation function, $O_i(s'_i, a_i, o'_i | \boldsymbol{\theta}_i)$, on $\boldsymbol{\theta}$. To make it represent the underlying $O_i(s', a_i, o'_i)$ correctly, we normalize its dimension of O_i. Next, we index the observation function with a_i and o'_i to obtain $\Pr(o'_i | s', a_i)$. Finally, we get the updated belief by weighting $\boldsymbol{B}'_S$ with $\Pr(o'_i | s', a_i)$:

$$\boldsymbol{B}_S = \sum_{a_i \in A_i} \sum_{o_i \in \Omega_i} \boldsymbol{B}'_S O_i(s, a_i, o_i | \boldsymbol{\theta}) \boldsymbol{v}_{a_i} \boldsymbol{v}_{o_i}$$

Thus, we finish a recurrence of the belief update. There are two routes for $\boldsymbol{B}_S$ to precede – one

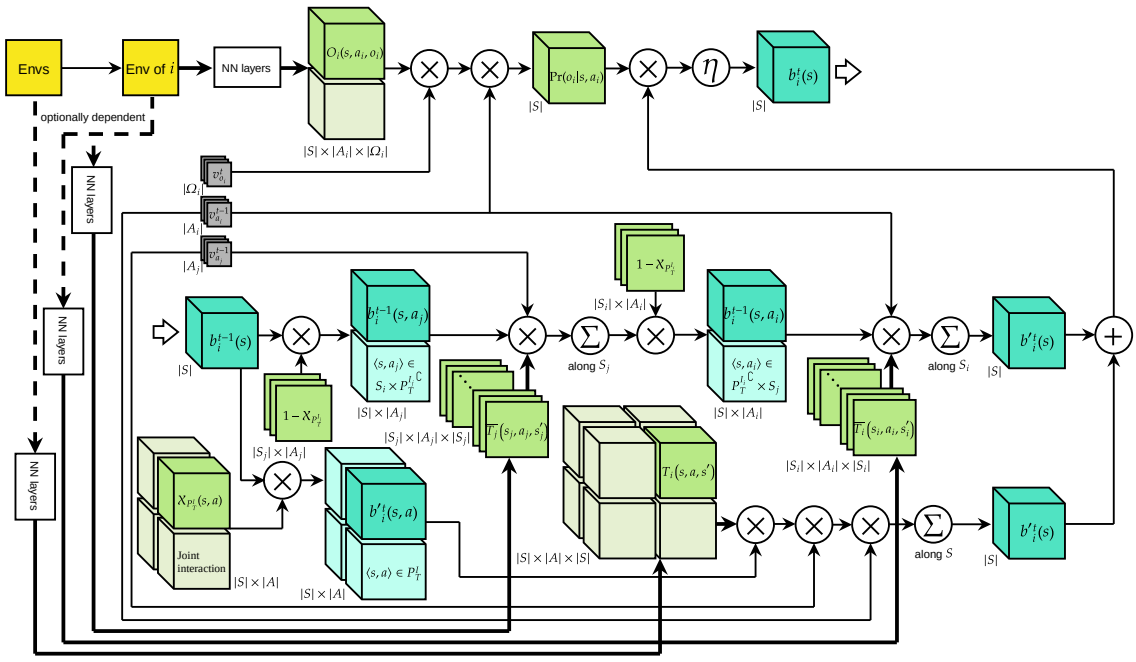


Figure 3.3: The belief update module of the sIPOMDPLite-net.

is to get back as the input for the next round of recurrence of the belief update, the other is to enter the QMDP planning module and to participate in the computation for the optimal policy.

3.2.4 Solution module

The solution module of sIPOMDPLite-net comprises two major submodules – the nested MDP planning module and the QMDP planning module. They share the same network architecture – the value iteration solver. In this subsection, we first present the design of the value iteration solver, expounding on its underlying logic that conforms to the value iteration in the framework of I-POMDP Lite with sparse interactions. Then, we show that the nested MDP planner and the QMDP planner employ the Q values, Q_i or Q_j, computed by the value iteration solver to obtain their respective policies.

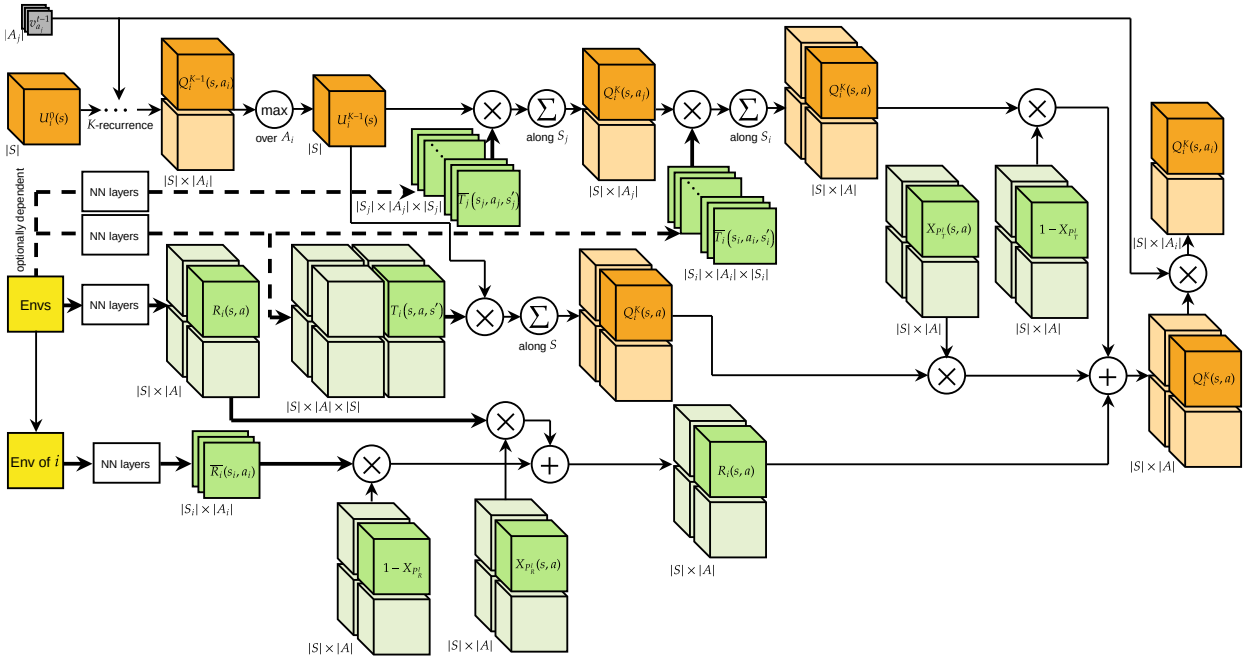


Figure 3.4: The value iteration solver.

Value iteration solver

The value iteration solver implements the NN analog for multiagent MDP value iteration based on the value function given in Eq. 3.5. Let us first consider the immediate reward. We condition the parameterized single-agent reward functions of agent i and agent j on $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$ while conditioning the multiagent parameterized reward functions of the two agents on entire input task parameter, $\boldsymbol{\theta}$. We set NN layers here to account for the dependencies, where they work to extract underlying reward patterns from $\boldsymbol{\theta}$.

The network architectural design for the long-term reward is similar to that for the belief update with actions, where we multiply $\boldsymbol{U}_S$ with $T_i(s, a, s'|\boldsymbol{\theta})$ for interactions and with $\overline{T}_i(s_i, a_i, s_i'|\boldsymbol{\theta}_i)$ and $\overline{T}_j(s_j, a_j, s_j'|\boldsymbol{\theta}_j)$ consecutively for non-interaction situations. Here in the value iteration module, we replace the belief tensor $\boldsymbol{B}_S$ with the state utility tensor $\boldsymbol{U}_S$. In the first round of the value iteration, we initiate $\boldsymbol{U}_S$ as a zero tensor based on Eq. 3.5 such that $Q_i^0(s, a) = R_i(s, a)$. The only difference

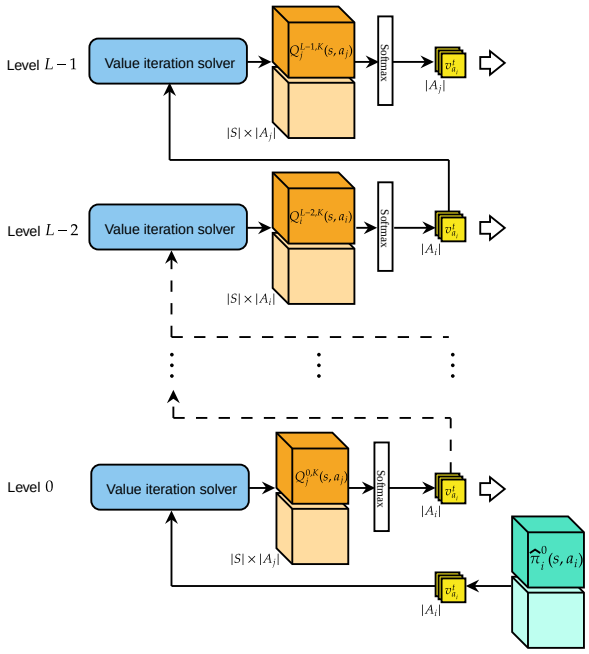


Figure 3.5: The nested MDP planning module generates other agents' mixed strategies by computing their Q values independently.

from the belief propagation is that in later recursions, we first multiply $\boldsymbol{U}_S$ first with the transition functions and then with the indicator functions. We have explained this in Subsection. 3.2.4.

We illustrate the architecture of the value iteration solver in Fig. 3.4.

Nested MDP planning module

The nested MDP planning module is an RNN with a hierarchical structure, which conforms to the setting of reasoning levels in the underlying nested MDP model. In this RNN, the hidden state is the policy, $\hat{\pi}_{i/j}^l$, output by each hierarchy. The most initial policy, which we denote by $\hat{\pi}^0$, is manually set to a uniform distribution over either A_i or A_j, depending on the top reasoning level L, for all $s \in S$. We sample action from this initial policy and input it to the 0th hierarchy of the planner to compute the level-1 policy. In the bottom hierarchy, $\hat{\pi}^0$ enters the embedded value iteration solver with the reward and transition functions. The value iteration solver outputs $Q^0(s,a)$ regarding joint

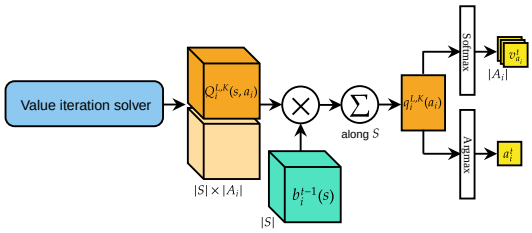


Figure 3.6: The QMDP planning module obtains the optimal policy for the subjective agent reasoning at the top level by weighting Q values with the current belief and then selecting the best move based on the maximum value in the result.

actions. To get the policy for a specific agent, according to Eq. 2.1, we need to weight $Q^0(s,a)$ by the opponent's policy $\hat{\pi}^0$ and sum it over the dimension of the opponent's actions. Thus, we get the Q values regarding only this agent's actions, on which we apply the Softmax operation along the dimension of actions to map Q values to the policy, i.e., $\hat{\pi}^1$. Continuing with such procedures, we sample action from $\hat{\pi}^1$ and input it to the level-1 planner and compute $\hat{\pi}^2$ till it reaches the highest hierarchy.

Hierarchies that account for a certain agent's policy share the same input models and the network structure. As agent i always reasons at the top level, L, where it applies the belief update and the QMDP solution, the highest level of the nested MDP module is $L-1$, where we solve j's MDP to predict its policy and offer it to the level L inference for i. Therefore, the $(L-1)$th hierarchy accepts R_j and actions sampled from $\hat{\pi_i}^{L-1}$, while the $(L-2)$th hierarchy receives R_i and actions from $\hat{\pi_j}^{L-2}$, and so on. All hierarchies share the convolutional filters representing transition functions, i.e., $\overline{T}_i$, $\overline{T}_j$, and T_i. we illustrate the general architecture of the nested MDP planning module in Fig. 3.5.

QMDP planning module

The QMDP planning module, which solves the subjective agent i's QMDP model, shares most of its architecture with a nested MDP hierarchy that solves i's MDP model. The only difference is

that we weight the computed $Q_i(s, a_i)$ by $b_i(s)$ output from the belief update module. Due to the imitation learning scheme which we choose for training the sIPOMDPLite-net, the output of the module, which is also the output of the network, should be the action in Softmax style. On the other hand, we directly output the action selected by $\arg\max_{a_i} \sum_s Q_i(s, a_i) b_i(s)$.

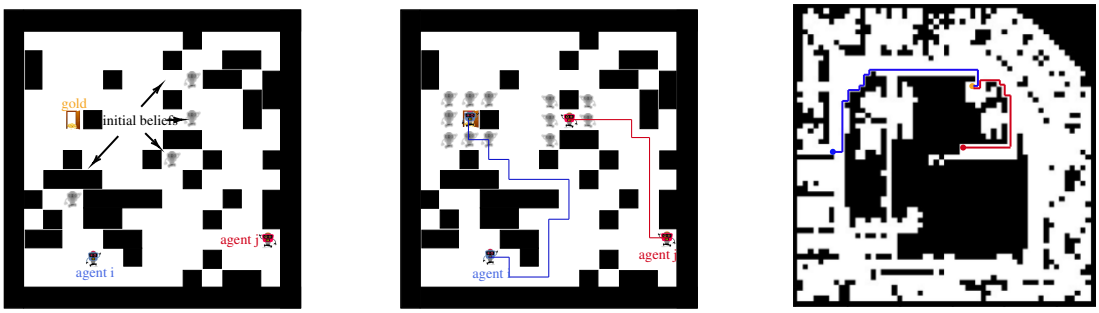
Chapter 4

Experiments

We elaborately design a handful of experimental domains exploiting multiagent properties to challenge our network. To study the benefits of explicitly encoding I-POMDP Lite models in the network and our network's overall genericity, we compare the performance of the learned policy and the expert policy on various unseen environments from different domains, including the larger and more complicated ones.

4.1 Experimental setup

We test the sIPOMDPLite-net on two domains—manually created Tiger-grids and real-world LIDAR maps. Each domain contains a parameterized set of tasks $\mathcal{W}(\Theta)$, where each $\boldsymbol{\theta} \in \Theta$ encodes a map for the environment, a goal, and an initial belief over each agent's state space. First, we randomly generate a sufficient number of samples of $\boldsymbol{\theta}$ for some small-scale environments. Then we construct the I-POMDP Lite with sparse interactions framework for the generated samples and apply the QMDP algorithm to solve it. Thus, we obtain the action and observation trajectories from the simulation and select those satisfying the preset success condition as an expert demonstration for later training.



(a) Initial state. (b) Stochastic state reset. (c) Laser maps.

Figure 4.1: Examples of two experimental domains. (a) The initial state of the task, where the subjective does not know exactly the locations of others and itself but maintain a belief. (b) A reset happening after an agent successfully seizes gold, where the system shifts each agent to a neighboring cell in either cardinal or intercardinal directions. (c) A down-sampled real-world LIDAR map can be regarded as an extension of the Tiger-grids, in which we mainly evaluate the network's ability of generalization.

We train a model on small environments and then evaluate it on larger ones. Furthermore, we apply it to real-world environments given by a set of LIDAR maps. Finally, we evaluate the network's overall performance by comparing it with its underlying framework. Below we briefly describe the experimental domains.

4.1.1 Two-agent Tiger-grid game

We introduce the Tiger-grid – a generalization of the classic Tiger problem to a grid world. Two robots act in a partially observable grid world filled with obstacles or free space in the problem. In each free cell, a door exists, either a pile of gold or nothing. Only one cell contains the gold, while all others are empty. We present such an environment in Fig 4.1a. Both agents have six actions: moving towards each of the four cardinal directions, listening to gain information, and opening the door. Only when the robot selects to listen will it get meaningful observations; otherwise, it will receive a random observation. We compress the observations into four binary values corresponding to obstacles in its four cardinal neighboring cells.

In the single-agent setting, an agent must navigate itself to circumvent obstacles present ran-

domly in cells and reach the cell where a pile of gold lies. Meanwhile, it only gets the gold by proactively executing the "open" action rather than simply being in the cell. When the agent chooses between staying and listening and opening the door in each cell, the problem becomes a Tiger problem. The action of listening costs -0.2; opening gains a reward of 10.0 if there is gold behind the door in the cell. Otherwise, it receives a -5.0 penalty. The action of moving integrates all individual Tiger problems happening in each cell and connects them with a navigation problem. Moving toward any direction has the identical cost of -0.5. Once the agent collides an obstacle, it receives a hurtful penalty of -50.0. The selection of gains and costs has to achieve a subtle balance. The agent should be neither too cautious and keep standing still nor too bold and moving recklessly without listening even if having collided with obstacles several times.

We make the Tiger-grid a multiagent system by imposing interference between involved agents. Specifically, both agents aim to seize the gold. Once an agent successfully achieves the goal, both of them are randomly relocated to a neighbor cell, specifically, a free cell in its cardinal and intercardinal directions, and the game continues. We illustrate a reset in Fig. 4.1b. This design ensures interactions between the agents. By modeling the other agent's intentions, a self-interested agent keeps the awareness of where the other is and when it will open the door, which helps the agent determine its current situation and plan for the future. In the Tiger-grid, the uncertainty regarding dynamics is rooted in the reset of agents' locations (4.1b). Hence, we keep the rest of the state transitions deterministic. We set the fault rate of observations to 0.01 independently in each direction.

When either agent consecutively seizes the gold three times or the trajectory length exceeds the limitation, the trajectory terminates. Thus, we regard a trajectory as successful if the subjective agent seizes the gold at least once.

We train a policy using demonstrated trajectories from 10,000 random 6×6 environments, 10 trajectories with different initial locations, initial beliefs, and gold positions for both agents from each environment. We then evaluate the trained model on separate sets of random environments

with gradually larger sizes, each with 500 environments. Finally, we repeat the evaluation for the trained policy on each task 100 times to compute the average values mitigating the impact of stochasticity during the simulation.

4.1.2 Real-world map navigation

We collect 2D LIDAR maps for realistic buildings' top-view layouts from the Robotics Data Set Repository and down-sample them to acceptable sizes. Then, to match the setup of input maps of Tiger-grid problems to evaluate the trained models directly, we map the value of pixels to binary values 1 and 0, corresponding to the free space or obstacles.

The two robots navigate in such processed "large" grids with all characteristics similar to Tiger-grids. Hence, the underlying configurations of sIPOMDPLite-net models are also the same. We illustrate an example for this domain in Fig. 4.1c.

Provided the model well-trained for 6×6 Tiger-grid tasks, we continue to train it on 10×10 and 12×12 tasks to generalize the environments better. Finally, We finalize the model trained for 12×12 and directly apply it to the down-sampled LIDAR maps.

4.2 Results and discussions

We display the experimental results for the two domains in Table 4.1 and Table 4.2, respectively. We present the mean and standard deviation of success rates, collision rates, and accumulated rewards of the learned policy and expert policy over 10 rounds of 100 repeats for each set of test environments.

sIPOMDPLite-net learns models that accurately approximate the underlying framework. The first objective of our approach is to learn models of the underlying framework accurately. We show this accuracy by visualizing the trained models. For the transition functions and the

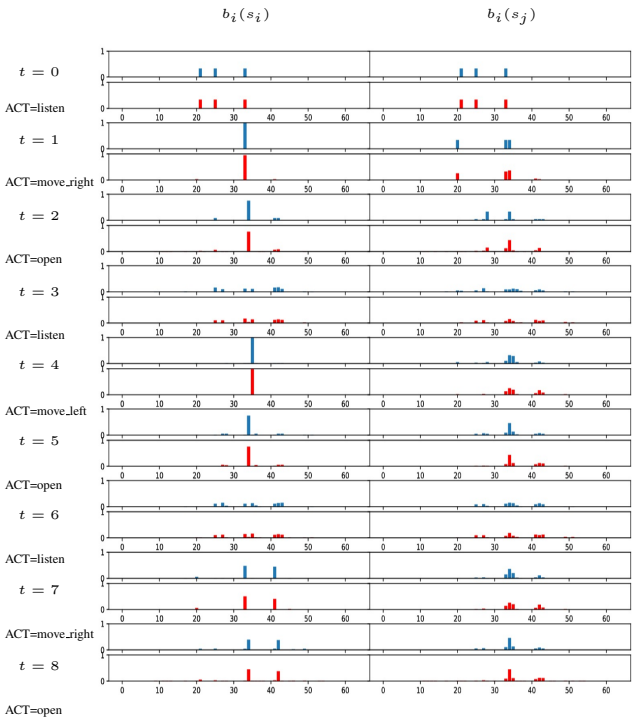


Figure 4.2: The visualization of agent i's belief update throughout a trajectory. The trajectory contains 8 time steps. For each time step, the first row illustrates the belief given by the underlying I-POMDP Lite framework, while the second row depicts that of the IPOMDPLite-net. For a better intuition, we present the beliefs over S_i and S_j on the left and right sides, respectively.

observation function, we picture an 8-step belief update given by both the underlying I-POMDP Lite with sparse interactions framework and our trained sIPOMDPLite-net in Fig. 4.2. Each row, consisting of two sub-rows, represents the two agents' beliefs after executing the action at the time step marked on the left side, where the sub-row in blue corresponds to the framework while that in red corresponds to the trained model. To make the chart easier to read, we portray $b_i(s_i)$ and $b_i(s_j)$ instead of $b_i(s)$, where $b_i(s_i) = \sum_{s_j} b_i(s)$ and $b_i(s_j) = \sum_{s_i} b_i(s)$. As such, the state space to visualize reduces from $|S| = |S_i| \times |S_j|$ to $|S_i| + |S_j|$. We see that every two sub-rows in the chart are mostly consistent, proving that our network accurately learns the underlying models of the framework via expert demonstrations.

Table 4.1: We evaluate the sIPOMDPLite-net's learned policy by comparing its performance on multiple sizes of Tiger-grid tasks with that of the I-POMDP Lite framework using QMDP solver. The Succ rate, F-open rate, and Colli rate represent the success rate, false open rate, and collision rate, respectively.

	I-POMDP Lite			**sIPOMDPLite-net**			
Env	Succ rate	F-open rate	Colli rate	Succ rate	Δ	F-open rate	Colli rate
6×6	0.858±0.003	0.208±0.002	0.059±0.002	0.851±0.003	0.004	0.235±0.003	0.070±0.003
7×7	0.812±0.003	0.192±0.002	0.056±0.002	0.804±0.003	0.009	0.215±0.003	0.085±0.002
8×8	0.820±0.003	0.188±0.002	0.066±0.003	0.790±0.004	0.037	0.204±0.003	0.098±0.004
10×10	0.712±0.004	0.150±0.002	0.076±0.003	0.731±0.003	0.027	0.172±0.004	0.115±0.003
12×12	0.694±0.004	0.118±0.002	0.040±0.004	0.743±0.004	0.071	0.150±0.004	0.102±0.003

sIPOMDPLite-net learns policies that generalize well to unseen tasks. The unseen tasks here refer to the set $\{W(\boldsymbol{\theta})|\boldsymbol{\theta} \in \boldsymbol{\Theta}^{\complement}_{train}\}$ as defined in 2.3. They share the same physical state space and agents' action space as those of the tasks in the training set but do not duplicate them for the specific obstacle distributions, the initial positions of the agents, and the goal locations. We train sIPOMDPLite-net with a set of 6×6 Tiger-grid tasks as $\{W(\boldsymbol{\theta})|\boldsymbol{\theta} \in \boldsymbol{\Theta}_{train}\}$ and evaluate it with the tasks sampled from $\{W(\boldsymbol{\theta})|\boldsymbol{\theta} \in \boldsymbol{\Theta}^{\complement}_{train}\}$. As the first row of Table. 4.1 suggests, the policy yields a comparable result to the expert's, with a similar success rate yet slightly higher collision and false-open rate. Thus, the performance is persuasive to show that our network learns a reasonably good policy to solve a specific set of tasks.

The model trained in relatively small and simple tasks by sIPOMDPLite-net applies to larger and more complex tasks without further training. Provided with the model trained for 6×6 Tiger-grid tasks, which represents a policy that solves such set of tasks, we evaluate its performance dealing with Tiger-grid tasks with larger environments, i.e., larger state space. The action space remains unchanged. Unlike so-called transfer learning, we do not train the model further for the new environments before the evaluation but directly evaluate it with them. As rows 2 to 5 of

Table 4.2: We compare the performance between the expert I-POMDP Lite policy and the policy learned by the sIPOMDPLite-net on four down-sampled LIDAR maps with respect to the success rate, the false open rate, and the collision rate.

	I-POMDP Lite			**sIPOMDPLite-net**			
Env	Succ rate	F-open rate	Colli rate	Succ rate	Δ	F-open rate	Colli rate
ACES	0.830±0.023	0.098±0.028	0.064±0.034	0.842±0.029	0.014	0.112±0.030	0.080±0.036
Fr-camp	0.820±0.017	0.125±0.030	0.040±0.031	0.803±0.025	0.021	0.150±0.036	0.048±0.034
Orebro	0.890±0.018	0.225±0.024	0.005±0.012	0.950±0.011	0.067	0.270±0.031	0.010±0.020
UW	0.940±0.021	0.172±0.022	0.032±0.012	0.910±0.017	0.032	0.188±0.041	0.020±0.015

Table 4.1 show, even with degradation of the performance as the size of the task environment increases, the policy given by sIPOMDPLite-net reaches the same level as that given by the expert. More surprisingly, our network performs even better than the expert in terms of the success rate only. Such transferability is more pronounced when dealing with much larger environments of the real-world LIDAR maps. Table 4.2 presents the sIPOMDPLite-net's performance compared with the expert's when the primitive model is further trained with 10×10 and 12×12 Tiger-grid tasks and evaluated on four LIDAR maps. The two perform comparably without a significant gap in all three statistics.

Appropriate prior knowledge regarding the underlying framework helps design the network architecture that yields better performance. It is intractable to train a network with a super complex internal logic and too many trainable variables. Therefore, eliminating part of the variables and simplifying the logic based on our prior knowledge regarding the hidden framework is rather important. For example, if we already know that the two agents share the same transition or reward function, we only need to train the network to learn one model instead of two. Another example is the observation function O_i. According to the setup of the Tiger-grid domain, an agent's observation depends on both states S and actions A_i, so theoretically, we should learn a tensor of

the shape $|S| \times |A_i| \times |\Omega_i|$ to represent O_i. However, only when the action is "stay and listen" the agent receives valuable observations; otherwise, the agent receives random observations from a uniform distribution over Ω_i. Thus, rather than learning the variable of shape $|S| \times |A_i| \times |\Omega_i|$, we learn one with shape $|S| \times |\Omega_i|$ and then stack it with another $(|A_i| - 1)$ constants with value $|\Omega|^{-1}$. Thus, we greatly reduce the number of variables to learn and guide the network to learn a more accurate model.

4.3 Ablation study

We carry out an ablation study on two small but representative test datasets to demonstrate the importance of sIPOMDPLite-net's critical components. We remove exactly one such component in every ablation experiment and train a model with the ablated architecture. We then study the role each component plays and to what extent it affects the network's performance.

Bayesian belief filter. In this experiment, we remove the Bayesian filter that recursively updates the subjective agent's beliefs to investigate its necessity. Without the belief update, the network keeps taking in the initial belief and completely leaves the policy searching duty to the QMDP planning module. The ablation essentially eliminates the network's recurrent property, the critical factor that the network can account for the belief update in the I-POMDP Lite planning. We

Table 4.3: Three ablation experiments for the 6×6 Tiger-grid games and the impact on the success rate, collision rate, and accumulated reward.

Ablation	Succ rate	F-open rate	Colli rate
sIPOMDPLite-net	**0.851±0.003**	**0.235±0.003**	**0.070±0.003**
sIPOMDPLite-net w/o belief update	0.001±0.000	0.001±0.000	0.115±0.000
sIPOMDPLite-net w/o nested MDP modeling	0.744±0.002	0.110±0.004	0.170±0.004
sIPOMDPLite-net w/o single-agent models	0.063±0.001	0.001±0.000	0.875±0.001

inspect the network's output action trajectories, finding that most of them are "stay and listen", which explains the low collision rate. Therefore, the Bayesian belief filter is indispensable in our network.

Nested MDP planner. In this experiment, we eliminate the nested MDP planner responsible for learning others' policies and predicting their actions to examine how considering others' intentions benefits the subjective agent in a multiagent system. The ablated network is essentially a QMDP-net. The second row of Table 4.3 shows that the ablated version receives worse results for all test items. This is due to the single-agent planning being unaware of the potential interference caused by the other agent. For instance, when the other agent takes the lead to seize the gold and reset underlying states, the subjective agent will not realize it and listen in time; instead, it may continue according to the previous belief. Although the agent might choose to observe at some time steps by learning the expert demonstration and fortunately pull itself back on track, it is not aware of the other's behaviors; hence it will not substantially help the agent accurately locate itself in a multiagent system. As such, the nested MDP planner is indeed crucial.

Single-agent models trained for non-interaction situations. In this experiment, we ablate all the single-agent models for both agents, including transition functions, i.e., $\overline{T_i}(s_i, a_i, s_i')$ and $\overline{T_j}(s_j, a_j, s_j)$, and reward functions, i.e., $\overline{R_i}(s_i, a_i)$ and $\overline{R_j}(s_j, a_j)$, while relying on the multiagent models, i.e., $T_i(s, a, s')$, $R_i(s, a)$, and $R_j(s, a)$, to take care of the training completely. In this case, the problem to solve is not necessarily under sparse interactions. However, according to the third row of Table 4.3, the trained multiagent models perform shockingly worse. This is due to the lack of guidance provided by the prior knowledge regarding the awareness of sparse interactions. With well-trained single-agent models that account for most of the inference, the multiagent models in training focus only on where interactions happen while ignoring the portion of non-interaction based on the attention mechanism of NNs. Therefore, the pre-trained single-agent models are

imperative in our network for reasonably good performance concerning sparse interactions.

4.4 Implementation details

This section details the implementation of our work mainly from two aspects – the generation of the training data, including the construction of task parameters and the creation of expert demonstrations, and the selection of specific network structures for some essential parts regarding a set of tasks represented by the Tiger-grid problems.

4.4.1 Task parameters and expert demonstrations

We initiate the construction of the Tiger-grid domain by randomly generating discrete grids. Each grid cell has a probability of $\Pr \sim U(0.1, 0.3)$ to be an obstacle. The two agents share the identical action and observation space, where $A = \{a \in N | 0 \leq a \leq 5\}$ and $\Omega = \{0, 1\}^4$. For each observation vector, an element of 1 represents the agent observes that the adjacent cell in the corresponding direction is free space, while an element of 0 means that it observes an obstacle in that cell. A common state of the domain consists of both agents' locations following a consistent order, while each agent takes grid cells as its private states, $S_i = S_j = \{s \in N | 0 \leq s \leq |M|^2\}$, where M is the side length of the grid. Hence, the common state space is the Cartesian product of each agent's private state space, $S = \{(s_i, s_j) | 0 \leq s_i \leq |M|^2, 0 \leq s_j \leq |M|^2\}$.

We then build underlying models for each grid environment. Since both agents' dynamics are local and spatial invariant, we use SciPy's sparse matrices to store the transition and reward functions. Thus, we can iteratively update beliefs through matrix multiplications and summations. Because an I-POMDP Lite approximately reduces to a POMDP and exactly solving a POMDP is expensive, we solve it with the more economic MA-QMDP and obtain a near-optimal policy. We implement the QMDP value iteration with the help of the MDP Toolbox, yielding the state utilities V and action values Q.

We use the described planning scheme to generate expert trajectories. The input to the framework includes the initial physical state, terminal state, and belief over the initial state. Both agents' initial actions are "stay and listen" by default. We save the output action and observation for each time step in an HDF5 database.

4.4.2 Network selection and design for problems with the spatial locality

For problems represented by the Tiger-grid, the states are defined in terms of spatial locations. If their actions can only lead to transitions from the current state to a subset of states within a certain range, we call such problems spatially local. We can employ CNNs into several vital parts of our network to solve such problems, including the belief propagation with actions and the long-term reward prediction in the value function, for an excellent approximation of their counterparts in the underlying framework.

Existing work [17, 19] has demonstrated the viability of applying 2D convolutions to single-agent spatially local problems, where they approximate the belief propagation and the calculation for expected Q values with a convolutional layer. The kernel of the layer plays a role as the transition function, which only captures the transition probabilities within the sliding window. The rationale of such approximation is that the operations with the transition function in the framework are consistent with the underlying logic of convolutions. Specifically, they all contain the dot product calculated by a set of multiplications and additions. A 2D convolution works as:

$$\mathcal{O}[n, m] = (\mathcal{I} * \mathcal{K})[n, m] = \sum_q \sum_p \mathcal{K}[q, p]\mathcal{I}[n - q, m - p]$$

where $\mathcal{I}$, $\mathcal{O}$, and $\mathcal{K}$ are the input, output, and kernel (convolutional filters), respectively. The input and the output are of the same shape $n \times m$, and the kernel is of the shape $q \times p$. In single-agent problems, there is $|S| = n \times m$. For each $s = (n_s, m_s) \in S$, a valid target state $s' = (n_{s'}, m_{s'})$ after taking any action satisfies $n_{s'} \in [n_s - r, n_s + r]$, $m_{s'} \in [m_s - r, m_s + r]$, where r is the step

length of an action. Thus, if $p = 2r + 1$ and $q = 2r + 1$, and we regard $\mathcal{I}$ as the current belief $b(s)$, $\mathcal{O}$ as the updated belief $b'(s')$, and $\mathcal{K}$ as the transition for a specific action a, the equation becomes exactly the single-agent belief update:

$$b'(s') = \sum_{s} T_a(s, s')b(s)$$

The fact above illustrates the consistency between the belief update with actions and the convolution. The same applies to the computation for expected Q values in the Bellman equation, where we swap the positions of s and s'.

We show that the applicability of using convolutions for approximation naturally carries over multiagent spatially local problems by expanding the dimensionality. Taking two-agent Tiger-grid problems as an example, we represent the common state space S as a 4D tensor, the Cartesian product of the two 2D tensors representing S_i and S_j. This representation maintains both agents' spatial localities as a whole.

Due to the overly computational intensity of high-dimensional convolutions, we only apply it for dealing with the interactions. As for non-interactive cases, we maintain the use of 2D convolutions. Hence, although the 4D kernel essentially represents the full transition function, we are only concerned about the part for interactions, not requiring equivalently accurate transitions to be learned for the non-interactive part, which benefits the training.

In the rest of the subsection, we demonstrate the specific usage of convolutions for the belief update and value iteration in sIPOMDPLite-net dealing with two-agent Tiger-grid problems and instantiate other important network structures that we mark in Fig. 4.3 and Fig. 4.4.

Belief update

In two-agent Tiger-grid problems, as stated in Subsection 4.4.1, each agent has a grid map in $\boldsymbol{\theta}$, which we define as their private state space, i.e., S_i and S_j. Provided that each grid map is a

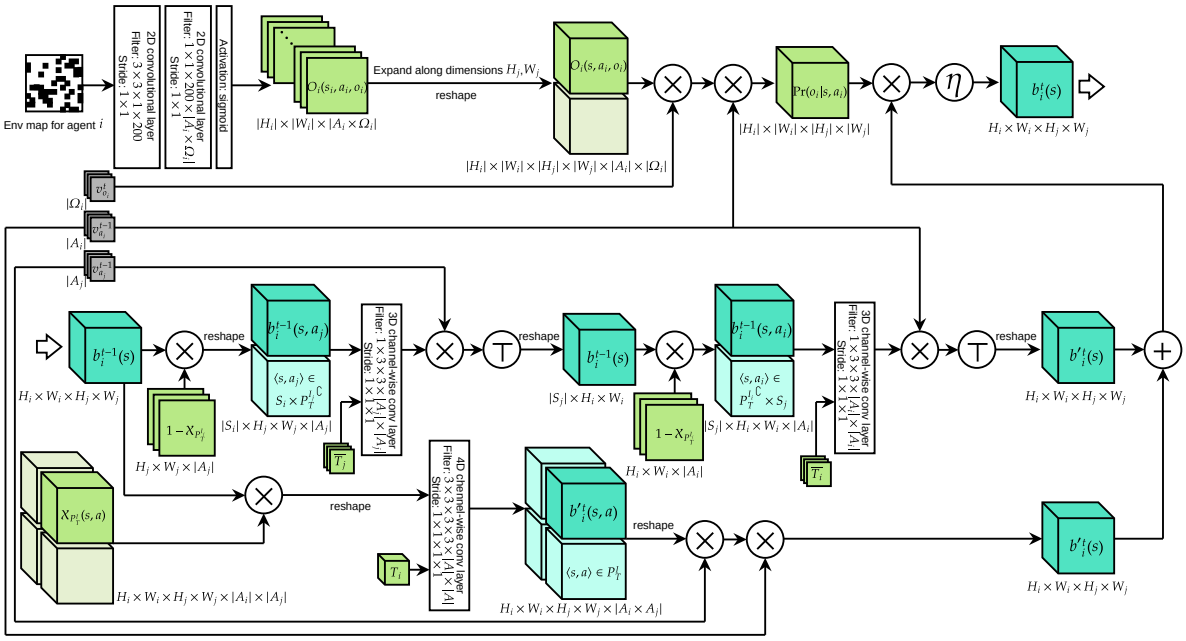


Figure 4.3: The specific architecture of sIPOMDPLite-net that designed for addressing problems with spatial locality like Tiger-grid problems, where we approximate transition functions with kernels for convolutional layers. We also employ a CNN to learn the observation function.

2D tensor, if agent i's map is of the shape $H_I \times W_i$, and agent j's is of $H_j \times W_j$, then we have $|S_i| = H_i \times W_i$ and $|S_j| = H_j \times W_j$. Hence, the common state space formed as a 4D tensor is of the shape $H_i \times W_i \times H_j \times W_j$, which is true for belief tensors.

As demonstrated in Subsection 3.2.3, we first update the non-interactive part of the input belief tensor $\boldsymbol{B}_S$, corresponding to $\langle s, a\rangle \in P_T^{I\complement}$. The resultant tensor after screening out the belief for $P_T^{I\complement}$ by the indicator tensor $1 - \boldsymbol{X}_{\boldsymbol{P_T^I}}$, denoted by $\boldsymbol{B}_{P_T^{I\complement}}$, is of the shape $H_i \times W_i \times H_j \times W_j \times A$, which we regard as a 4D image with $|A|$ channels. Provided that we should apply a two-step update here with the agents single-agent transition functions, T_j and T_i, in order, and that the single-agent update is approximated by 2D convolutions, we need to first reshape $\boldsymbol{B}_{P_T^{I\complement}}$ to match the 2D convolution with kernel $\mathcal{K}_{\overline{T_j}}(s_j, a_j|\boldsymbol{\theta})$ and then transpose the result to match another 2D convolution with kernel $\mathcal{K}_{\overline{T_i}}(s_i, a_i|\boldsymbol{\theta})$.

Here is a trick for the 2D convolutions. As we know, when a convolution happens, the kernel slides over the $H \times W$ plane while being fixed in the channel dimension. We require our network to be extendable in both S_i and S_j, so if we reshape $\boldsymbol{B}_{P_T^I\mathtt{C}}$ to some $H_j \times W_j \times |S_i \times A_j|$, where S_i and A_j are merged in channels, we cannot transfer what learn to tasks with different size of environments. The solution is to reshape $\boldsymbol{B}_{P_T^I\mathtt{C}}$ to $|S_i| \times H_j \times W_j \times A_j$, a 3D image with S_i depths and A_j channels. A 3D kernel can thus slides along S_i, H_i, and W_i, and fixes only A_j. By setting $\mathcal{K}_{\overline{T_j}}(s_j, a_j|\boldsymbol{\theta})$'s and $\mathcal{K}_{\overline{T_i}}(s_i, a_i|\boldsymbol{\theta})$'s depth and stride along S_i to 1, we essentially conduct the 2D convolution merely in the form of a 3D one.

Continuing with this line, we address the two-step update for $\boldsymbol{B}_{P_T^I\mathtt{C}}$. When it comes to the update for $\boldsymbol{B}_{P_T^I}$, we initialize $\mathcal{K}_{T_i}(s, a|\boldsymbol{\theta})$ and apply it directly to the 4D convolution. After both updates, we select the channels corresponding to the given a_i and a_j and then sum them up, which yields the updated belief tensor $\boldsymbol{B}'_S$.

When dealing with Tiger-grid problems, we learn agent i's observation O_i with a CNN, which captures the information of local environments for each $s_i \in S_i$ that matches the observations, specifically, the distribution of obstacles within a given range centered on s_i, and maps it to a valid representation of $\Pr(o_i|s_i, a_i)$ by forcing the weights on dimension $o_i \in \Omega_i$ to sum to 1.

Next, to match $\boldsymbol{B}'_S$ for the correction, we expand O_i on dimensions H_j and W_j and tile the probabilities for the expanded dimensions. Finally, we get the fully updated belief tensor $\boldsymbol{B}_S$ by multiplying $\boldsymbol{B}'_S$ and the expanded O_i and normalizing the product over S.

We illustrate the details of convolutional layers in Fig. 4.3.

Value function

The application of convolutions in the value iteration module is similar to that in the belief update. We still represent the two-agent transition function with a 4D kernel and represent the single-agent ones with 3D kernels of the depth 1.

As for immediate rewards, we also learn them via CNNs. In Tiger-grids, if considering only the

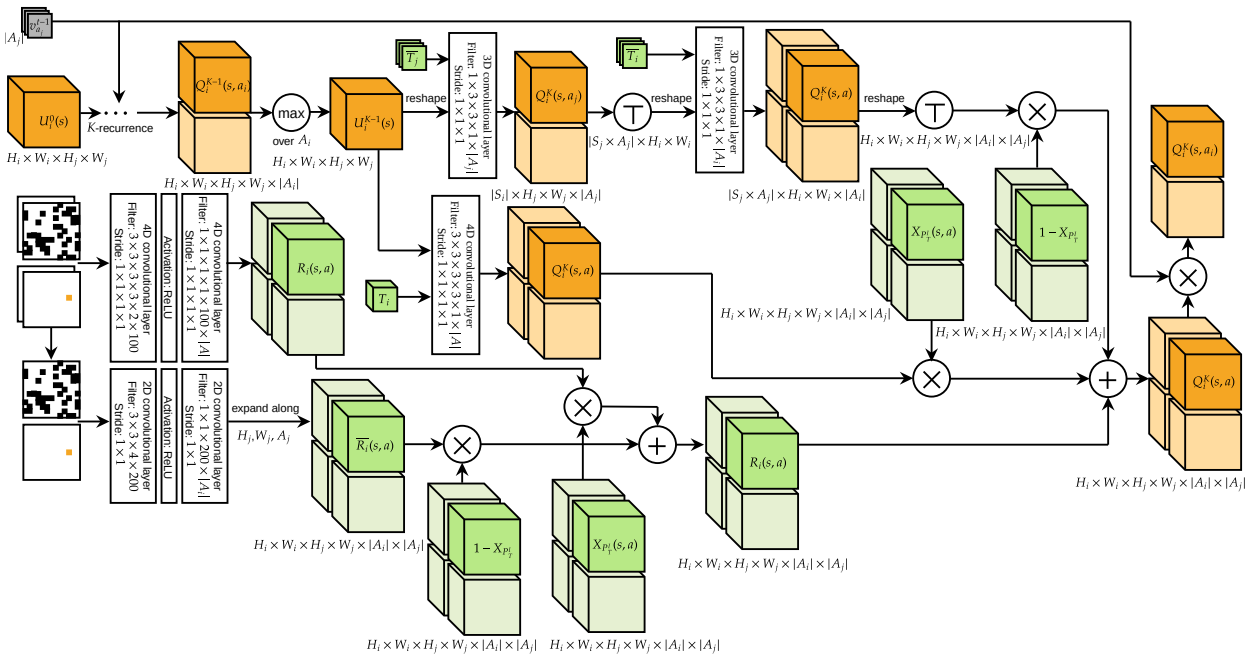


Figure 4.4: Value iteration solver of sIPOMDPLite-net that to deal with spatially local problems such as Tiger-grid problems, where we approximate transition functions with kernels for convolutional layers and learn agents' reward functions with CNNs.

non-interaction situations, the immediate rewards for an agent depend on whether it opens the door in a Gold cell or not and whether it collides an obstacle or not. Hence, we condition the rewards on the grid map and goal map of the agent given in $\boldsymbol{\theta}$. We stack them together as a two-channel image and feed it into a CNN, where a kernel of the same shape as the transition function is applied. We learn single-agent reward functions, including $\overline{R_i}(s_i, a_i)$ and $\overline{R_j}(s_j, a_j)$, in this way.

When it comes to the two-agent rewards, such as $R_i(s, a)$ and $R_j(s, a)$, where we must consider common states and joint actions, it is no longer sufficient to condition them on single-agent grid maps and goal maps. Instead, we merge the agents' respective grid maps and goal maps to compute the Cartesian product, hence getting a grid map and a goal map in the 4D space. Then, still stacking them together, we conduct the 4D convolution for several layers, mapping the two-channel image to a tensor that represents the two-agent reward function with $|A| = |A_i| \times |A_j|$ channels, each

representing the reward of executing the corresponding joint action a for all $s \in S$.

With all required immediate reward functions and the reward interaction indicators, we can get the true two-agent reward functions as in Eq. 3.5. We illustrate the details of CNNs in Fig. 4.4.

4.5 Training details

In Subsection 4.4.1, we have shown the procedures of creating expert trajectories and storing them in the database. Below we introduce how these trajectories are processed for mini-batch training and imitation learning.

4.5.1 Training with expert demonstrations

Since we only choose successful expert trajectories to be our demonstration, we first filter out failed ones. Then, we produce mini-batches for backpropagation through time (BPTT), which is widely used for training recurrent NN (RNN). We break down a full trajectory into several sub-trajectories with the length equal to the backpropagation step size, each of which is wrapped by a block. We set the step size to 5 for 6×6 and 7×7 Tiger-grids and to 8 when training the model further on 8×8, 10×10, and 12×12 grids. Provided with the total number of blocks needed, we get the number of steps for each epoch by dividing it using the mini-batch size. We set the mini-batch size to 100 for all of our experiments. Next, we joint trajectories end to end in blocks for all mini-batches. Once a trajectory terminates, we pad it until reaching the block limitation. New trajectories begin from the next block even if other trajectories in the batch have not terminated yet. We still need to distinguish if a sub-trajectory is the start of its original trajectory. This determines whether we should assign the initial belief to it or not. As such, we complete creating batch samples. The final step of yielding training data is to extract actions and observations from stored steps of all trajectories and specify any pair of consecutive actions as the input and target (or label) action, respectively. Hence, we construct the mapping from an input action-observation

pair to the corresponding next action in the demonstration.

To apply the imitation learning, we intuitively define the loss as the cross-entropy between the network's predicted actions and the demonstrated actions along the expert trajectories, as it shows in Eq. 4.1.

$$J(\boldsymbol{\theta}) = -\frac{1}{N}\frac{1}{|A_i|}\sum_{t=T}^{T+N-1}\sum_{a_i^t \in A_i} \log \Pr_{\pi_i(\boldsymbol{\theta})}\left(a_i^t = \tau_{a_i}^t\right) \tag{4.1}$$

We apply RMSProp optimizer with 0.9 decay rate and 0 momentum. The learning rate is 1×10^{-3} for training from scratch and 1×10^{-5} for further training a trained model with new data. We combine early stopping with patience and exponential learning rate decay for the adaptive gradient descent. Specifically, we set the initial patience to 30 epochs and the rest to 10 epochs; and we perform 20 iterations of learning rate decay in total. It means that we do not decay the learning rate at first until the loss does not decrease for 40 consecutive epochs on the validation set. Subsequently, we decay the current learning rate by 0.9 if the loss does not decrease for 15 epochs with it. We set the ratio of the training set and validation set to $9:1$.

Another essential hyperparameter for our network is the number of value iterations. In the underlying planning framework, the iteration terminates when it converges to the true state utilities $\|U_{i+1} - U_i\| < \epsilon(1-\gamma)/\gamma$, where U_i and U_{i+1} are utilities of the ith and $(i+1)$th iteration, ϵ is the maximum error acceptable for convergence, and γ is the discounted factor.

However, in our network, we execute the value iteration for a specific number of steps given by the hyperparameter K. We train the network with 6×6 Tiger-grid domain with randomly generated environments and then directly apply the trained model to tasks with larger environments, i.e., larger state space. We select K for a set of tasks based on empirical trials. We first select a benchmark value for it, where $K = 4N$. Then, we search for the best K within the range of $[4N - 10, 4N + 10]$ in units of 5 by evaluating the policy trained with each value of K. We pick the one with the best performance to further evaluate the policy on larger tasks. We list the values

selected for K regarding each N in 4.4.

Table 4.4 presents all the important hyperparameters that we discussed above.

Table 4.4: Essential hyperparameters.

Hyperparameter	Argument
Step size ($N = 6$)	5
Step size ($N = 10, 12$)	6
Mini-batch size	50
Initial learning rate (from scratch)	0.001
Initial learning rate (re-training)	0.0001
Maximum epochs	1000
Train-valid ratio	9 : 1
K ($N = 6$)	24
K ($N = 7$)	30
K ($N = 8$)	32
K ($N = 10$)	40
K ($N = 12$)	50

4.5.2 Transferring pre-learned knowledge

As demonstrated in 3.2, we train the network to learn transition functions including $T_i(s_i, a_i)$, $T_j(s_j, a_j)$, and $T_i(s, a)$, reward functions including $R_i(s_i, a_i)$, $R_i(s, a)$, $R_j(s_j, a_j)$, and $R_j(s, a)$, and the observation function $O_i(s, a_i, o_i)$. Learning these models from scratch simultaneously via backpropagation is prohibitively difficult. We alleviate this dilemma by dividing the training procedure into multiple steps, learning part of the models at first and then using them as priors to continue training the remaining models. Given that agents mostly follow their single-agent models in tasks, we can first learn them based on single-agent demonstrations. We actually train several distinct QMDP-nets and obtain $T_i(s_i, a_i)$, $T_j(s_j, a_j)$, $R_i(s_i, a_i)$, $R_j(s_j, a_j)$, and $O_i(s_i, a_i, o_i)$. After that, we freeze the weights of these learned models while focusing on training the multiagent models for interactions. In this way, we guide the network training in the right direction, which is another sense of effective use of a priori knowledge.

Chapter 5

Conclusion

We presented the sIPOMDPLite-net, a deep recurrent policy network explicitly encoding the I-POMDP Lite models for self-interested planning with sparse interactions in multiagent systems. The network is flexible and computationally economic compared with existing multiagent planning networks, such as IPOMDP-net.

Performance of sIPOMDPLite-net reaches the comparable level of that given by the underlying I-POMDP Lite policies in the majority of tasks from various experimental environments. Moreover, our trained model even outperforms the expert demonstrations in a few tasks. Empirical results show that the learned policies generalize well to unseen environments and larger and more sophisticated tasks.

We will continue our research in the following directions in the future.

First, we set the rounds of recurrence for the value iteration, i.e., K, as a hyperparameter responsible for a whole set of parameterized tasks. Practically, Nevertheless, the selection of K should depend on the horizon length of each specific task rather than general environment sizes. One probable solution may be incorporating K as a trainable variable and training the network in a meta optimization manner.

Second, providing indicator functions as priors in task parameters is a relatively strong assump-

tion for our network. However, learning it through neural networks, which optimize the model by the conventional backpropagation and gradient descent, is intractable due to the binary values of the indicators. Besides, introducing additional variables to train will make the training harder to converge to an optimal policy. Therefore, searching for an appropriate representation for the indicator functions in the network is a critical challenge for us.

Third, the setup for states is inflexible. Even for basic Tiger-grid problems, when the number of single-agent states increases, the number of common states increases more sharply. This fact severely limits the scalability of our network. Therefore, we are interested in coming up with a more proper way to represent the states in a dynamic way, where we can greatly reduce the number of states, which in accordance decreases the size of beliefs, utilities, and models conditioned on the inputs before we begin the I-POMDP Lite inference.

References

[1] E. A. Hansen, D. S. Bernstein, and S. Zilberstein, "Dynamic programming for partially observable stochastic games," in *AAAI*, vol. 4, pp. 709–715, 2004.

[2] J. Hu, M. P. Wellman, *et al.*, "Multiagent reinforcement learning: theoretical framework and an algorithm.," in *ICML*, vol. 98, pp. 242–250, Citeseer, 1998.

[3] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International Conference on Machine Learning*, pp. 4295–4304, PMLR, 2018.

[4] Y. Yang, R. Luo, M. Li, M. Zhou, W. Zhang, and J. Wang, "Mean field multi-agent reinforcement learning," in *International Conference on Machine Learning*, pp. 5571–5580, PMLR, 2018.

[5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[6] P. J. Gmytrasiewicz and P. Doshi, "A framework for sequential planning in multi-agent settings," *Journal of Artificial Intelligence Research*, vol. 24, pp. 49–79, 2005.

[7] B. Rathnasabapathy, P. Doshi, and P. Gmytrasiewicz, "Exact solutions of interactive pomdps using behavioral equivalence," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 1025–1032, 2006.

[8] P. Doshi and P. J. Gmytrasiewicz, "Monte carlo sampling methods for approximating interactive pomdps," *Journal of Artificial Intelligence Research*, vol. 34, pp. 297–337, 2009.

[9] P. Doshi and D. Perez, "Generalized point based value iteration for interactive pomdps.," in *AAAI*, pp. 63–68, 2008.

[10] B. Ng, C. Meyers, K. Boakye, and J. J. Nitao, "Towards applying interactive pomdps to real-world adversary modeling.," in *IAAI*, 2010.

[11] T. N. Hoang and K. H. Low, "Interactive pomdp lite: Towards practical planning to predict and exploit intentions for interacting with self-interested agents," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, p. 2298–2305, AAAI Press, 2013.

[12] F. S. Melo and M. Veloso, "Decentralized mdps with sparse interactions," *Artificial Intelligence*, vol. 175, no. 11, pp. 1757–1789, 2011.

[13] F. S. Melo and M. Veloso, "Heuristic planning for decentralized mdps with sparse interactions," in *Distributed Autonomous Robotic Systems*, pp. 329–343, Springer, 2013.

[14] C. Yu, M. Zhang, and F. Ren, "Coordinated learning by exploiting sparse interaction in multiagent systems," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 51–70, 2014.

[15] C. Yu, M. Zhang, F. Ren, and G. Tan, "Multiagent learning of coordination in loosely coupled multiagent systems," *IEEE transactions on cybernetics*, vol. 45, no. 12, pp. 2853–2867, 2015.

[16] Y. Hu, Y. Gao, and B. An, "Learning in multi-agent systems with sparse interactions by knowledge transfer and game abstraction.," in *AAMAS*, pp. 753–761, 2015.

[17] A. Tamar, Y. WU, G. Thomas, S. Levine, and P. Abbeel, "Value iteration networks," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[18] S. Niu, S. Chen, H. Guo, C. Targonski, M. Smith, and J. Kovačević, "Generalized value iteration networks:life beyond lattices," 2018.

[19] P. Karkus, D. Hsu, and W. Lee, "Qmdp-net: Deep learning for planning under partial observability," 12 2017.

[20] Y. Han and P. Gmytrasiewicz, "Ipomdp-net: A deep neural network for partially observable multi-agent planning using interactive pomdps," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 6062–6069, 2019.

[21] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, "Benchmarking model-based reinforcement learning," *arXiv preprint arXiv:1907.02057*, 2019.

[22] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, *et al.*, "Model-based reinforcement learning for atari," *arXiv preprint arXiv:1903.00374*, 2019.

[23] A. Ayoub, Z. Jia, C. Szepesvari, M. Wang, and L. Yang, "Model-based reinforcement learning with value-targeted regression," in *International Conference on Machine Learning*, pp. 463–474, PMLR, 2020.

[24] R. Jonschkowski, D. Rastogi, and O. Brock, "Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors," 2018.

[25] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, “Learning policies for partially observable environments: Scaling up,” in *ICML*, 1995.

[26] J. Pineau, *Tractable planning under uncertainty: exploiting structure*. Carnegie Mellon University, 2004.

Appendix A

More Explanation of The Net Architecture

A.1 Kronecker indicator converter

It is intuitive to interpret the equations using the set theory. Let us consider S_i and A_i as two sets, and the set of its entire state-action combinations is the Cartesian product of the two, i.e., $S_i \times A_i$. Continuing with this line, for the set of i's interaction-triggering state-action pairs, denoted by P^{I_i}, there is $P^{I_i} \subseteq S_i \times A_i$. Similarly, we get agent j's state-action space, $S_j \times A_j$, and its active interaction triggering state-action pairs $P^{I_j} \subseteq S_j \times A_j$. As introduced in 1.1, for the problems we discuss in this thesis, the following equations always hold: $S = S_i \times S_j$, $A = A_i \times A_j$. Hence, $S \times A = (S_i \times A_i) \times (S_j \times A_j)$, which we regard as the universal set. We can accordingly derive the set of joint interaction-triggering state-action pairs as:

$$
\begin{aligned}
P^I &= \left[P^{I_i} \times (S_j \times A_j)\right] \cup \left[(S_i \times A_i) \times P^{I_j}\right] \\
&= \left(\left[P^{I_i} \times (S_j \times A_j)\right] \setminus \left[P^{I_i} \times P^{I_j}\right]\right) \cup \left(\left[(S_i \times A_i) \times P^{I_j}\right] \setminus \left[P^{I_i} \times P^{I_j}\right]\right) \cup \left(P^{I_i} \times P^{I_j}\right)
\end{aligned}
$$

So far, we have clearly explained the derivation of agents' common state-action pairs for interactions given each of their private ones. Hence, we can naturally transform the derivation to

account for the binary-value indicator functions as in Subsection 3.1.1. We know that the indicators work as indexing the interactive state-action pairs, where they assign 1 to such pairs and 0 to others. Thus, we replace single-agent universal sets $S_i \times A_i$ and $S_j \times A_j$ with all-ones tensor $\boldsymbol{J}_i$ and $\boldsymbol{J}_j$, replace the subsets corresponding to each agent's single-agent interactive state-action pairs, P^{I_i} and P^{I_j} with their single-agent indicators, $\boldsymbol{X}_{P^{I_i}}$ and $\boldsymbol{X}_{P^{I_j}}$, replace Cartesian products with Kronecker products, and replace set unions and subtractions with arithmetic sums and subtractions. To generalize the above, we have:

$$\begin{aligned}\boldsymbol{X}_{P_R^I} &= \boldsymbol{X}_{P_R^{I_i}} \otimes \boldsymbol{J}_j + \boldsymbol{J}_i \otimes \boldsymbol{X}_{P_R^{I_j}} - \boldsymbol{X}_{P_R^{I_i}} \otimes \boldsymbol{X}_{P_R^{I_j}} \\ \boldsymbol{X}_{P_T^I} &= \boldsymbol{X}_{P_T^{I_i}} \otimes \boldsymbol{J}_j + \boldsymbol{J}_i \otimes \boldsymbol{X}_{P_T^{I_j}} - \boldsymbol{X}_{P_T^{I_i}} \otimes \boldsymbol{X}_{P_T^{I_j}}\end{aligned} \tag{A.1}$$

Eq. A.1 is the macroscopic counterpart for Eq. 3.3, based on which we design the indicator transformer module of our network in Section 3.2.

A.2 Belief update module

In our elaboration of the belief update module in Subsection 3.2.3, we mentioned the input belief tensor $\boldsymbol{B}_S$ and the output belief tensor $\boldsymbol{B}'_S$. The subscript S means the belief tensor is of the shape $|S|$ and represents the belief for $s \in S$. Different subscripts for $\boldsymbol{B}$ represent beliefs for different sets.

Consistent with the framework, the first part of the belief update module works to update the $\boldsymbol{B}_S$ with a_i and a_j. Following Eq. 3.4, for the interactive part, we apply an element-wise multiplication to filter the belief over interactions from $\boldsymbol{B}_S$ with the transition interaction indicator, $\boldsymbol{X}_{P_T^I}$. Such multiplication broadcasts $\boldsymbol{B}_S$'s shape from $|S|$ to $|S| \times |A|$. Then, we update $\boldsymbol{B}_S$ with $T_i(s, a, s'|\boldsymbol{\theta})$, a trainable variable approximating $T_i(s, a, s')$. Based on the principle of learning as accurate a model as possible, we enforce the NN analog to inherit properties of its underlying counterpart. In the case of $T_i(s, a, s'|\boldsymbol{\theta})$, we normalize it over the dimension of s'. The network

structure simulating the update with actions generally implements the following operation:

$$\boldsymbol{B}'_{P_T^I} = \sum_{s \in S} \sum_{a_i \in A_i} \sum_{a_j \in A_j} \boldsymbol{B}_{P_T^I} T_i(s, a, s' | \boldsymbol{\theta}) \boldsymbol{v}_{a_i} \boldsymbol{v}_{a_j}$$

where $\boldsymbol{v}_{a_i}$ and $\boldsymbol{v}_{a_j}$ are indexing vectors for a_i and a_j.

As the equation shows, we update the belief for each $a \in A$, while we only concern about $\boldsymbol{B}'_{P_T^I}$ corresponding to the given $a = \langle a_i, a_j \rangle$. The specific class of network structure with which $T_i(s, a, s' | \boldsymbol{\theta})$ is trained depends on the domain it deals with. Section 4.4.2 demonstrates an instance of solving Tiger-grid problems with spatial locality, where we encode the belief propagation into a convolutional layer with $T_i(s, a, s' | \boldsymbol{\theta})$ as the kernel.

On the other hand, we execute the two-step belief propagation for non-interaction situations. Similar to the $T_i(s, a, s' | \boldsymbol{\theta})$, we create $\overline{T_i}(s_i, a_i, s'_i | \boldsymbol{\theta}_i)$ and $\overline{T_j}(s_j, a_j, s'_j | \boldsymbol{\theta}_j)$ to approximate underlying $\overline{T_i}(s_i, a_i, s'_i)$ and $\overline{T_j}(s_j, a_j, s'_j)$. Based on Eq. 3.4, we first multiply $\boldsymbol{B}_S$ with $1 - \boldsymbol{X}_{P_T^{I_j}}$. However, the former is of the shape $|S|$ while the latter is of $|S_j| \times |A_j|$. To make the dimensions match for the multiplication, we reshape $\boldsymbol{B}_S$ to $|S_i| \times |S_j|$. Then we multiply the product with $\overline{T_j}(s_j, a_j, s'_j | \boldsymbol{\theta}_j)$ and pick the $\boldsymbol{B}_S$ corresponding to the given a_j. In the second step, we duplicate such procedures with $1 - \boldsymbol{X}_{P_T^{I_i}}$ and $\overline{T_i}(s_i, a_i, s'_i | \boldsymbol{\theta}_i)$.

The next step of the belief update, i.e., the correction, has been illustrated in Subsection 3.2.3.

A.3 Value iteration solver

The value iteration solver implements the NN analog for multiagent MDP value iteration based on the value function given in Eq. 3.5. Let us first consider the immediate reward. We define the NN counterparts for $R_i(s, a)$, $R_j(s, a)$, $\overline{R_i}(s_i, a_i)$ and $\overline{R_j}(s_j, a_j)$ as $R_i(s, a | \boldsymbol{\theta})$, $R_j(s, a | \boldsymbol{\theta})$, $\overline{R_i}(s_i, a_i | \boldsymbol{\theta_i})$, and $\overline{R_j}(s_j, a_j | \boldsymbol{\theta_j})$, respectively. We condition $\overline{R_i}(s_i, a_i | \boldsymbol{\theta_i})$ and $\overline{R_j}(s_j, a_j | \boldsymbol{\theta_j})$ on their own portions in $\boldsymbol{\theta}$ while conditioning $R_i(s, a | \boldsymbol{\theta})$ and $R_j(s, a | \boldsymbol{\theta})$ on entire $\boldsymbol{\theta}$. Since for any given $s \in S$ and

$a \in A$, there are $R_i(s,a) \in \mathbb{R}$ and $R_j(s,a) \in \mathbb{R}$, which is also true for $\overline{R_i}(s_i,a_i)$ and $\overline{R_j}(s_j,a_j)$, we do not explicitly regulate what the network learns for the reward functions.

Following Eq. 3.5, for the part of interactions, we first update $\boldsymbol{U}'_S$ with $T_i(s,a,s'|\boldsymbol{\theta})$, which leads to $\boldsymbol{Q}_S$. we then apply the element-wise multiplication to filter Q values over interactions from $\boldsymbol{Q}_S$ with the transition interaction indicator, $\boldsymbol{X}_{P_T^I}$. The resulting value tensor is the long-term reward in the Bellman equation.

On the other hand, we deal with the two-step value function for non-interaction situations. Based on Eq. 3.5, we first multiply $\boldsymbol{U}'_S$ with $\overline{T_j}(s_j,a_j,s'_j|\boldsymbol{\theta}_j)$ and $\overline{T_i}(s_i,a_i,s'_i|\boldsymbol{\theta}_i)$ consecutively. Then, we multiply the resulting Q value tensor with $1-\boldsymbol{X}_{P_T^I}$. Thus, we get the Q values for the non-interaction part, $\boldsymbol{Q}_{P_T^I\complement}$. By summing up $\boldsymbol{Q}_{P_T^I}$ and $\boldsymbol{Q}_{P_T^I\complement}$, we obtain the Q value tensor $\boldsymbol{Q}_S$ for one round of value iteration.

Next, we add the parameterized immediate reward, i.e., $R_i(s,a|\boldsymbol{\theta})$, to $\boldsymbol{Q}_S$. After that, different from the belief update, we select the Q values matching $a_j \in A_j$, where we get agent i's Q value tensor of the shape $|S| \times |A_i|$ while compute maximum over A_i. The network structure representing the update with actions generally implements the following operation:

$$\boldsymbol{U}_S = \max_{a_i \in A_i} \sum_{a_j \in A_j} \left\{ \sum_{s' \in S} \left[\boldsymbol{U}'_S T_i(s,a,s'|\boldsymbol{\theta}) \boldsymbol{X}_{P_T^I} + \boldsymbol{U}'_S \overline{T_j}(s_j,a_j,s'_j|\boldsymbol{\theta}_j) \overline{T_i}(s_i,a,s'|\boldsymbol{\theta}_i)(1-\boldsymbol{X}_{P_T^I}) \right] + R_i(s,a) \right\} \boldsymbol{v}_{a_j}$$