Neurocomputing 399 (2020) 449–466

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

XCS with opponent modelling for concurrent reinforcement learners

Hao Chen, Chang Wang, Jian Huang*, Jiangtao Kong, Hanqiang Deng

College of Intelligence Science and Technology, National University of Defense Technology, Changsha, Hunan, 410073, China

ARTICLE INFO

Article history:
Received 16 July 2019
Revised 26 December 2019
Accepted 24 February 2020
Available online 12 March 2020

Communicated by Dr. W. Gao

Keywords:
Opponent modelling
XCS
Markov games
Reinforcement learning

ABSTRACT

Reinforcement learning (RL) of optimal policies against an opponent agent also with learning capability is still challenging in Markov games. A variety of algorithms have been proposed for solving this problem such as the traditional Q-learning-based RL (QbRL) algorithms as well as the state-of-the-art neural-network-based RL (NNbRL) algorithms. However, the QbRL approaches have poor generalization capability for complex problems with non-stationary opponents, while the learned policies by NNbRL algorithms are lack of explainability and transparency. In this paper, we propose an algorithm X-OMQ(λ) that integrates eXtended Classifier System (XCS) with opponent modelling for concurrent reinforcement learners in zero-sum Markov Games. The algorithm can learn general, accurate, and interpretable action selection rules and allow policy optimization using the genetic algorithm (GA). Besides, the X-OMQ(λ) agent optimizes the established opponent's model while simultaneously learning to select actions in a goal-directed manner. In addition, we use the eligibility trace mechanism to further speed up the learning process. In the reinforcement component, not only the classifiers in the action set are updated, but other relevant classifiers are also updated in a certain proportion. We demonstrate the performance of the proposed algorithm in the hunter prey problem and two adversarial soccer scenarios where the opponent is allowed to learn with several benchmark QbRL and NNbRL algorithms. The results show that our method has similar learning performance with the NNbRL algorithms while our method requires no prior knowledge of the opponent or the environment. Moreover, the learned action selection rules are also interpretable while having generalization capability.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

How to represent and store value functions and/or policies is one of the greatest challenges in reinforcement learning (RL) applications [1]. A RL agent learns through trial-and-error interactions with its environment which can be typically modeled as Markov Decision Processes (MDPs). Although the learning results of Q-learning-based RL (QbRL) algorithms are easy to understand, they do not have generalization capability and thus difficult to apply to complex problems. For example, the QbRL algorithms maintain a record of all state-action pairs in a table, which requires full exploration of the sate-action spaces. As the complexity of the problem increases, the state-action space grows exponentially with the number of state and action variables, and the exploration process can be very time-consuming [2].

Compared with single-agent RL, multi-agent reinforcement learning (MARL) is more difficult for non-stationary MDPs. In an example of robot soccer game, there are several concurrent learning agents that the policies of the opponents are unavailable and the learning is based on the observation of the opponents' behaviors. If the opponents are modeled as part of the environment, the state transition function and the reward function are non-stationary and changing over time [3]. In addition, the interdependence between the agents makes it even more difficult to find the optimal policy because an opponent is also evolving its own policy concurrently. Furthermore, the state-action space grows dramatically with the number of agents, which makes the problem of the curse of dimensionality more severe in MARL than in single agent RL [4,5].

To address the learning problems in multi-agent scenarios, Littman has formulated the framework of Markov Games and proposed the Minimax-Q algorithm [6] for fully competitive tasks. Unlike MDPs, the optimal policy is sometimes non-deterministic (i.e., probabilistic) in Markov Games [6] because of the interdependence between the agents. For games with alternating turns, the agent knows the action taken by others in advance, and the non-deterministic optimal policy becomes deterministic [6,7]. In this paper, we focus on the alternating fully competitive tasks, in which the players perform actions alternately and learn concurrently.

Trying to improve the learning performance in zero-sum Markov Games (ZSMG), several improved algorithms have been

* Corresponding author.
E-mail address: nudtjhuang@hotmail.com (J. Huang).

https://doi.org/10.1016/j.neucom.2020.02.118
0925-2312/© 2020 Elsevier B.V. All rights reserved.

proposed based on the Minimax-Q algorithm through spatial, temporal, or action generalization, such as Minimax-QS [2], Minimax-Q(λ) [8], Minimax-SARSA [8], Heuristically-accelerated Minimax-Q (HAMMQ) [7,9], etc. Minimax-QS and HAMMQ speed up convergence through domain-related prior knowledge, and their performance mainly depends on the choice of the spreading function [2,10] or heuristics [7,9], respectively. However, the prior knowledge or the heuristic is not always available if the opponent is learning concurrently. Besides, as QbRL algorithms, they directly store learned policies as Q-tables, which might be infeasible when handling complex problems. Furthermore, Minimax-Q like algorithms use the concept of equilibria by assuming that the opponent always follows the optimal action selection policy, while Minimax-Q follows a conservative policy. However, the opponent may follow a suboptimal policy (not always choosing the best action) or other kinds of policies rather than the optimal policy. Therefore, the opponent's behavior model has to be taken into account for learning a corresponding action selection policy.

Models of other players can be obtained by analyzing the history data of the observed behaviors and the consequent rewards. US-L* [11] formulates the interactions among the players as a repeated game where the agent adapts its strategy according to the history of the game, while the opponent is represented by a deterministic finite automaton. Hyper-Q [12] learns the values of strategies and actions by taking other agents' model into the state-action value representation. Besides, Uther et al. [13] and Weinberg et al. [14] have introduced other players' model into dynamic tasks by calculating the frequencies of actions taken by other agents in every state. However, due to the introduction of other players' model, the state-action space of the learning agent are further expanded and this may lead to a difficult optimal strategy search process. Similar ideas can also be found in neural-network-based RL (NNbRL) algorithms, such as multi-agent deep deterministic policy gradient (MADDPG) [15] and deep Bayesian policy reuse (deep BPR+) [16], which update other players' models with sampled observations.

One efficient way of handling large state-action space is to leverage the idea of generalization, by which we can find a good approximate solution using limited computational resources. This is achieved by trying to generalize from the examples of the desired function (e.g., a state-action value function) to form an approximation of the whole problem. The neural network has been used as a function approximator to get generalized learning results by updating the connection weights in all the layers. Besides, a serial of NNbRL algorithms [15,17–21] has been proposed. However, such generalization makes the learning results more difficult to interpret and understand [1].

The eXtended Classifier System (XCS) [22,23] is an accuracy based Learning Classifier System (LCS) [24,25]. In XCS, the RL part is responsible for payoff prediction and the genetic algorithms (GA) [26] part is responsible for evolving the population of action selection rules. XCS can evolve accurate, maximally general, and interpretable action selection rules with reasonable time. In the literature, XCS has been used in both cooperative RL tasks [27–29] and competitive RL tasks [30,31]. However, these algorithms treat the XCS agents independently without integrating other players' model into the representation of the XCS agents. In this paper, we take advantage of the mechanism of XCS to learn the action selection policy with opponent modelling.

The main contributions of the paper are as follows:

- We propose a concurrent policy learning algorithm X-OMQ(λ) for ZSMG which can not only search for the optimal policy by evolving a fixed-size population of action selection rules, but the learned generalized action selection policy is interpretable.
- We construct the opponent's model from scratch based on the observations of the opponent's states and actions while the model is simultaneously used to select actions and improve the action selection policy.
- We design a new mechanism to update the relevant classifiers in the population by matching the situation and agent's action with the historical data stored in the trace set, thus speeding up the policy learning.

We have evaluated the performance of the proposed algorithm in three environments, i.e., the Littman's soccer domain [6], the hunter prey problem [32], and the Hexcer environment [13].

The rest of the paper is organized as follows. Section 2 reviews the related work of RL in Markov Games and XCS in multi-agent system (MAS). Section 3 briefly introduces the main mechanisms of XCS, followed by the details of the proposed algorithm in Section 4. Section 5 describes the experiments and discusses the results of the experiments. Finally, Section 6 concludes the paper and outlines our plans for future work.

2. Related work

2.1. Reinforcement learning in Markov Games

Markov Games [6] are extension of single-agent MDPs to multi-agent scenarios including ZSMG and general-sum Markov Games. In ZSMG, there are two players (the agent and the opponent) with opposed goals, where one player tries to maximize the target function while the other player tries to minimize it. In contrast, general-sum Markov Games support more than two players, without the need for opposed target functions. In this paper, we focus on the problem of ZSMG. Markov Games can be formulated as a tuple $\langle n, S, A_1, \ldots, A_n, P, r_1, \ldots, r_n \rangle$, where:

- ▹ n is the number of agents;
- ▹ S is a set of finite states of the agents;
- ▹ A_i is the action set of the player i;
- ▹ P is a state transition function, P: $S \times A_1 \ldots \times A_n \mapsto \Pi(S)$, which describes the probability of changing from the current state s to the next state s' when the players execute $a_1, \ldots, a_n$, respectively;
- ▹ r_i is the reward function of the player i, r_i: $S \times A_1 \ldots \times A_n \mapsto R$.

In this paper, we discuss the zero-sum games between two players, i.e., $n = 2$. Denote by A_1 the action set of the RL agent, and A_2 the action set of the opponent.

Littman proposed the Minimax-Q learning algorithm [6] to solve ZSMG, which combines the Minimax algorithm in games and Q-learning [33] in RL. It is similar to Q-learning except that the *max* operator is replaced by the *minimax* operator. The next state and the reward are determined by the agent's action a in A_1 and the opponent's action o in A_2. The Q-values of Minimax-Q are updated as follows:

$$Q(s,a,o) \leftarrow Q(s,a,o) + \alpha\left[r(s,a,o) + \gamma V\left(s'\right) - Q(s,a,o)\right] \tag{1}$$

where α is the learning rate, $V(s)$ is the state function, and s' is the state of the next time step. For deterministic action policies [6], $V(s)$ is calculated as follows:

$$V(s) = \max_{a \in A_1} \min_{o \in A_2} Q(s,a,o) \tag{2}$$

Action selection strategies such as $\epsilon - greedy$ and Boltzmann distribution [1] can also be used in Minimax-Q, and the optimal policy is calculated as follows:

$$\pi^*(s) = \arg\max_{a \in A_1} \min_{o \in A_2} Q^*(s,a,o) \tag{3}$$

As a result, the Minimax-Q agent selects action conservatively (i.e., assuming the opponent always chooses the best action). In this paper, we use the RL algorithm name plus the agent to represent that the agent uses this algorithm for policy learning. Similar to Q-learning, Minimax-Q converges to the Nash equilibrium (NE) when the joint state-action pairs are visited infinitely [34]. But it is not guaranteed for Minimax-Q, because the agent's next state depends on the opponent's action in the given state. If the opponent never performs some actions in given states, it can be difficult for the agent to acquire the optimal policy. Besides, adopting the single-step updating rule, Minimax-Q has a low learning efficiency and almost randomly selects actions in the early stage of learning.

Win or learn fast (WoLF) [35] and Generalized Infinitesimal Gradient Ascent WoLF (GIGA-WoLF) [36] are examples of the direct policy search MARL algorithms. The heuristic of WoLF is to learn quickly when the agents perform poorly and learn cautiously when they perform well. It is proved that for a restrict class of iterated matrix games, it converges to a rational policy by altering the learning rates according to the WoLF principle. WoLF policy-hill climbing (WoLF-PHC) [35] and policy dynamics based WoLF (PD-WoLF) [37] are heuristic algorithms using WoLF rules in dynamic tasks.

There can be multiple NEs in a game and the policy converges to a NE may not be the optimal for MARL agent. Therefore, the goal of MARL has been changed to obtain the optimal payoff in the presence of other agents rather than to consider NE as a criterion. Based on this idea, agent-tracing methods have been proposed to construct other agent's models in both fully cooperative tasks [38] and fully competitive tasks [13]. Hyper-Q [12] incorporates other agents' models into the state vector. AWESOME (Adapt When Everybody is Stationary, Otherwise Move to Equilibrium) [39] has been proposed for repeated matrix games to play against a stationary opponent. Weinberg et al. [14] have introduced non-stationary converging policies (NSCP) to general-sum Markov Games to get the best response policy by inferring the model of non-stationary opponents. These algorithms combine other agents' model with the state-action value functions. However, with the increase of the complexity of the environment and the number of agents, the computational complexity grows exponentially, resulting in the dimensional curse problem.

The huge and/or continuous state space can be addressed by introducing the neural network as the function approximator into RL. Mnih et al. [18] have presented an asynchronous framework for deep RL and proposed the asynchronous advantage actor-critic (A3C) for continuous control problems. In order to learn efficiently in rich and efficiently, Heess et al. [20] have proposed the distributed proximal policy optimization (DPPO) algorithm, in which data collections are distributed over workers. Similar ideas can be found in proximal policy optimization (PPO) [21]. However, the efficiency improvement of these NNbRL algorithms requires the participation of multiple parallel learners and therefore requires more computing resources.

To further improve the learning efficiency, other player's behaviors have been introduced into NNbRL algorithms. Lowe et al. [15] have extended deep deterministic policy gradient (DDPG) [19] to MAS and proposed MADDPG for multi-agent cooperation and competition. Raileanu et al. [40] have proposed self other-modeling (SOM) to infer the other players' hidden goals from their behaviors and use these estimates to choose actions. However, SOM requires longer training time and assumes other players' policies are stationary. He et al. [41] have proposed to incorporate mix-of-experts architecture to deep Q-network (DQN) [17] for different types of opponents and presented the deep reinforcement opponent network (DRON). One limitation of DRON is that it cannot response optimally against a particular type of opponent. To address this, Hao et al. [16] have proposed to extend BPR+ with DQN and incorporate other players' models into the agent's belief model for accurate opponent policy detection and optimal response policy reuse. However, it assumes that the opponent cannot change its policy within an episode. Moreover, these NNbRL algorithms use neural networks as the function approximator and the learned results are tuned parameters that lack interpretability. In other words, it is difficult to understand the process by which the algorithms output the selected action through state inputs in the application phase.

2.2. Approaches to accelerate Minimax-Q learning

The Minimax-Q agent updates only one Q-value during each learning process which is not efficient. Minimax-Q(λ) [8] combines the mechanism of eligibility trace [1] with Minimax-Q to speed up the learning. Eligibility trace records the state-action trajectory that the agent has visited. When updating the Q-table, not only the current state value is updated, the previously visited (s, a, o) tuples in the trajectory are also updated.

Minimax-SARSA [8] is an on-policy algorithm, which combines Minimax-Q and SARSA [1]. The updating rule is similar to (1) except that $V(s')$ is replaced by the actual joint action value:

$$V\left(s'\right) = Q\left(s', a', o'\right) \tag{4}$$

where a' in A_1 and o' in A_2 represent the actions of the next time step of the agent and the opponent, respectively. Minimax-SARSA(λ) [8] combines Minimax-SARSA with eligibility trace, using the actual action value from the next state to update the current state. The performance strongly depends on the actual learning policy. As eligibility trace is used to update the state-action values previously visited, Minimax-Q(λ) and Minimax-SARSA(λ) are regarded to accelerate learning through temporal generalization.

Minimax-QS [2] introduces the spreading function [10] to Minimax-Q, in which a single experience can update a couple of state-action pairs through spatial generalization. The spreading function $\sigma_t(s, a, o, s_i, a_i, o_i) \in [0, 1]$ defines the similarity among state-action pairs. In [2], $\sigma_t(s, a, o, s_i, a_i, o_i) = g_t(s, s_i)\delta(a, a_i)\delta(o, o_i)$, where g_t is the state similarity function, and δ is the Kronecker delta function. For each update, the Q-value of the tuple (s_i, a_i, o_i) is updated simultaneously according to the similarity degree to the tuple (s, a, o). The updating rule of Minimax-QS is as follows:

$$\begin{aligned} Q(s_i, a_i, o_i) \leftarrow\ & Q(s_i, a_i, o_i) + \alpha\sigma(s, a, o, s_i, a_i, o_i) \\ & \times\left[r(s, a, o) + \gamma V\left(s'\right) - Q(s_i, a_i, o_i)\right] \end{aligned} \tag{5}$$

where $V(s)$ is the same as (2). If the spreading mechanism vanishes at least as quickly as the learning rate α, then Minimax-QS converges to optimal values. However, its performance highly depends on the accuracy of the domain-related spreading function.

HAMMQ [7,9] uses a heuristic function to guide action selection, which also requires the domain-related prior knowledge. Compared with Minimax-Q, HAMMQ only changes the policy of action selection without the way of updating Q-values. For deterministic action policy, the action selection rule is:

$$\pi(s) = \begin{cases} \arg\max\limits_{a\in A_1} \min\limits_{o\in A_2} \left[Q(s, a, o) + \xi H_t(s, a, o)^{\beta}\right] & \text{if } prob \geq \varepsilon \\ a_{random} \quad \text{otherwise} \end{cases} \tag{6}$$

where ξ and β are real values, H is the heuristic function, and $prob$ is a random variable drawn from a unifrom distribution over [0, 1]. For a given state s in S and opponent action o in A_2, H is represented as follows:

$$H(s, a, o) = \begin{cases} \max\limits_{a_i\in A_1} Q(s, a_i, o) - Q(s, a, o) + \eta & \text{if } a = \pi_H(s) \\ 0 \quad \text{otherwise} \end{cases} \tag{7}$$

where η is a small real value, $\pi_H(s)$ is the heuristic policy in state s. Compared with Minimax-QS, HAMMQ needs more precise domain-related prior knowledge to construct a valid heuristic function. However, it is not easy to obtain a handcrafted heuristic function in practical application.

2.3. Application of XCS in multi-agent systems

As an accuracy based LCS, XCS has been widely used in MAS such as social simulation [42,43], agent modelling [44,45], financial [27,46], path planning [29], market [47], distributed learning [48], etc.

For competitive tasks, Inoue et al. [30] have compared accuracy-based XCS with strength-based XCS in a simplified soccer domain. Accuracy-based XCS has performed better in a multi-agent environment with a teammate without the problem of over generalization on classifiers. Lode et al. [31] have introduced XCS to the non-observable MDPs, in which each agent is modeled as a single, independent XCS.

For cooperative tasks, Li et al. [27] have integrated XCS and neural networks to make reasonable investment decisions and reduce man-made errors or mistakes. Énée et al. [28] have defined the minimal requirements needed by a multi-agent classifier system to evolve communication for a fully cooperative task. Chen et al. [29] have presented the DH-HML approach to assign priority dynamically to resolve conflicts among agents with shared resources.

However, most of these algorithms has regard XCS agents as independent individuals in MAS, without considering the interdependent relationships of agents within the XCS framework. In other words, there are few studies on how to integrate other agents' models into the representation of XCS and the learning of optimal action selection policy.

3. The eXtended classifier system

LCS is a rule based evolutionary learning framework, in which the agent learns to perform a task by evolving a population [P] of condition-action rules through temporal difference (TD) learning and GA. We note that such a rule is also called a classifier *cl* in the LCS literature. In this paper, we use the term classifier and rule alternatively as needed.

XCS is an accuracy based Michigan-style LCS, in which the strength of the classifier is replaced by the payoff prediction p, the prediction error ϵ, and the fitness F. Besides, each classifier *cl* also keeps additional parameters to support learning in XCS, e.g., the action set size *as* estimates the averaged size of the action set that contains *cl*, the time stamp *ts* indicates the time-step of the last GA in the action set that contains *cl*, the experience exp specifies the number of times *cl* is selected in an action set, etc.

The overview of XCS is shown in Fig. 1. When XCS receives an input situation s from the environment, a match set [M] is formed, containing every classifier in [P] whose condition matches s. The *covering* mechanism is necessary if the number of actions in [M] is less than a predefined threshold θ_{mna}, which guarantees the candidate actions for selection are sufficient. Then, new classifiers with conditions matched with s will be created for those missing actions with predefined predicted payoff p along with ϵ and F. The *system prediction* of action is a fitness-weighted average of the predicted payoff of all classifiers in [M] advocating that action. The prediction array in Fig. 1 contains the *system predictions* of all candidate actions. Moreover, the action with the highest *system prediction* is selected to execute and the classifiers in [M] advocating that action forms the action set [A]. After the selected action is performed and the actual reward is obtained, the prediction error is calculated and used to update all the relevant rules in the previous time step's action set $[A]_{-1}$.

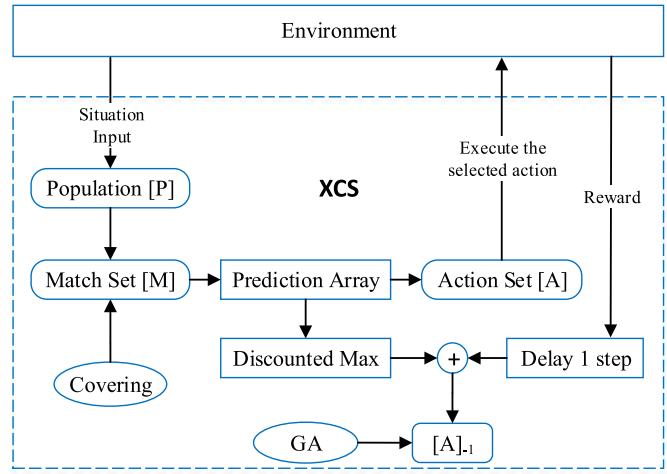


Fig. 1. An overview of XCS.

GA is responsible for classifier generalization in LCS. In traditional LCS, GA in the match set may produce inaccurate classifiers by crossing parents with different actions [23]. For this reason, XCS executes GA in the action set (named niche GA). Specifically, niche GA is triggered if the average time step of classifiers in the action set exceeds the threshold θ_{GA}:

$$\frac{\sum_{cl\in[A]}(t - cl.ts)\cdot cl.num}{\sum_{cl\in[A]} cl.num} > \theta_{GA} \tag{8}$$

where [A] indicates the action set, *cl.num* indicates how many identical classifiers exist in the population, and t is the current time step. In this paper, we use the dot to refer to the parameter in a classifier. Two parent classifiers are selected from [A], with probability proportionate to their fitness. Then, the offspring classifiers which can still match the current input are created through the crossover and mutation operators. Finally, parameters of the offspring classifiers are initiated. The numerosity *num* and experience exp are initialized to 1, and the payoff prediction p, the prediction error ϵ, and the fitness F are initialized in a certain proportion of the parents.

Another major difference between XCS and the traditional LCS is the use of *macroclassifier* and the *subsumption* mechanism. After new classifiers are created through GA or classifiers in the action set are updated, the *subsumption* mechanism is triggered. *GA subsumption* checks the offspring classifiers to see whether their conditions can be subsumed by the classifier that is accurate enough ($cl.\epsilon < \epsilon_0$) and sufficiently experienced ($cl.exp > \theta_{sub}$). *Action set subsumption* finds the most general classifier that is accurate and sufficiently experienced in the action set and tries to subsume other classifiers. After applying the *subsumption* mechanism, the numerosity parameter *num* of the subsumer (named *macroclassifier*) is increased. In other words, one *macroclassifier* represents multiple regular classifiers with the same parameters. *Macroclasifier* and *subsumption* are two major mechanisms to speed up matching classifiers with an input and evolving maximally general classifiers.

In order to maintain a fixed-size population, the classifiers with low fitness should be deleted. Besides, in order to ensure population diversity, the number of classifiers with different actions in the population should be approximately the same. For these purposes, sufficiently experienced ($cl.exp > \theta_{del}$) classifiers with low average fitness ($cl.F/cl.num < \delta \cdot F_{average}$), large action set size, and large numerosity are more likely to be deleted.

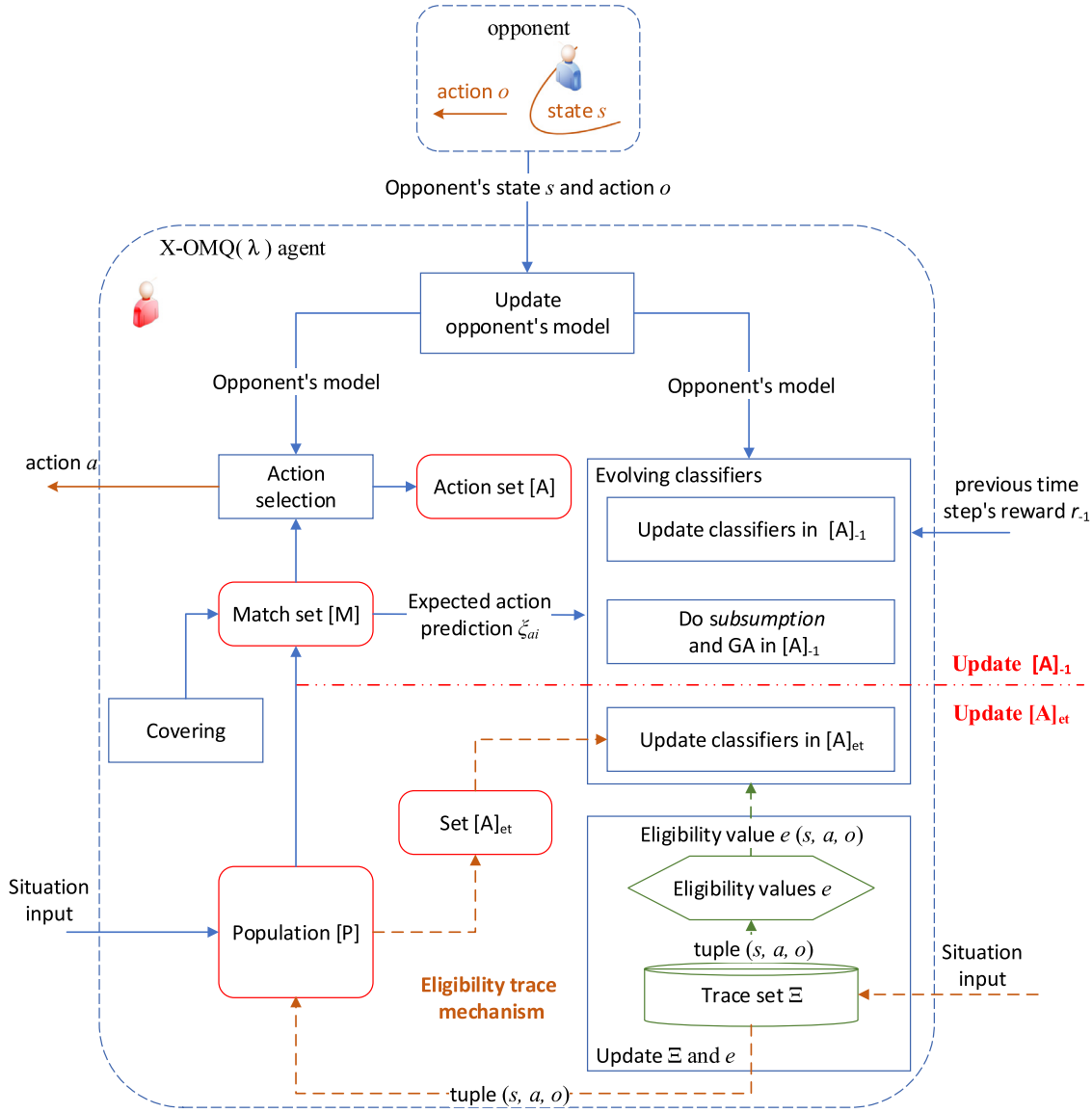


Fig. 2. An overview of X-OMQ(λ). Data modules containing classifiers are represented by the red rounded rectangles, and the blue rectangles describe the main function modules. The classifier update process consists of two parts. First, as shown in the upper part and the solid arrow lines of the figure, the classifiers in [P] match the current situation are updated in $[A]_{-1}$. Second, the classifiers in [P] that can match the historical information recorded in the trace set Ξ are reinforced in $[A]_{et}$, as shown in the lower part and the dashed arrow lines of the figure. Note that the opponent's model is used not only to select action but evolve classifiers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

4. The proposed X-OMQ(λ) algorithm

In this section, we propose a novel XCS-based algorithm by combining with opponent modelling for concurrent reinforcement learning in ZSMG. In addition, we incorporate the eligibility trace mechanism to speed up learning. We named our proposed algorithm X-OMQ(λ) because it is the extended version of XCS to ZSMG combing opponent modelling. Moreover, the incorporated updating mechanism of the eligibility trace is similar to the Q(λ) algorithm. We focus on deterministic policies and assume that the XCS-based agent can observe the state and the action of the opponent. The overview of X-OMQ(λ) is shown in Fig. 2. The X-OMQ(λ) agent constructs the opponent's model from scratch and updates it by observing the opponent's states and actions. Besides, the opponent's model is used for action selection and evolving classifiers. In the learning part, not only the classifiers in the previous time step's action set are adjusted, but the classifiers in the population that can match with the trace set are also updated.

4.1. Opponent modelling

As depicted in Section 2.1 and 2.2, the Minimax-Q agent assumes that the opponent always follows an optimal policy and therefore selects action conservatively. Minimax-QS and HAMMQ enhance the learning process through spatial and action generalization. We note that these heuristic-based algorithms must ensure the correctness of the prior knowledge. However, accurate prior knowledge is not always available. In contrast, we construct the opponent's model from scratch only based on the behaviors of the opponent, requiring no prior knowledge. Then, the opponent's model is updated on-line while being used for action selection.

Similar to NSCP [14] that extends the modelling method in JAL [38] to dynamic tasks, X-OMQ(λ) maintains a table K to record

the number of times that action o has been executed by the opponent in the state s. Then, the estimated action distribution of the opponent in s can be calculated as follows:

$$\sigma(s, o_i) = \frac{K(s, o_i)}{\sum_{o \in A_2} K(s, o)} \tag{9}$$

4.2. Representation and creation of classifiers

We incorporate the opponent's model into the classifiers of X-OMQ(λ), which helps the agent make the best response in given states. A classifier in X-OMQ(λ) maps a condition c and an action a to a payoff prediction vector $\boldsymbol{p}$ with an associated prediction error ϵ and a fitness value F as follows:

$$\langle c : a \rangle \rightarrow \langle \boldsymbol{p}, \epsilon, F \rangle \tag{10}$$

note that $\boldsymbol{p}$ is a vector rather than a value as in XCS. An element $p[o]$ in $\boldsymbol{p}$ denotes the predicted payoff obtained by the X-OMQ(λ) agent if the agent executes the action a while the opponent selects a corresponding action o. In other words, the dimension of $\boldsymbol{p}$ is the same with the cardinal of the opponent's action set A_2.

The condition c in the classifier cl is represented by a vector of the ternary alphabet $\{0, 1, \#\}$ in the same way as XCS, where the wildcard # can be either 0 or 1. Besides, the situation s (i.e., the state in RL) relies on binary coding and thus each bit of the situation s is encoded with 0 or 1. When the X-OMQ(λ) agent receives an input situation s from the environment, a match set [M] is formed, containing every classifier in [P] whose condition matches s. This is achieved by checking whether the situation s matches the condition in the classifiers. We say that the condition c matches s only if all specific components (i.e., not a #) are the same in all corresponding positions. For example, the condition 1011# matches the situation 10, 111.

In the early stage of training, the reward from the environment can be very sparse, e.g., the agent is rewarded only when it reaches the goal state. As a result, the elements in the predicted payoff vector $\boldsymbol{p}$ and the prediction error ϵ of the classifiers used for action selection will be zero. According to the mechanism of XCS, the fitness F of these classifiers should be increased because they can predict the reward accurately. However, these classifiers do not contribute to the learning, and they have to be temporally kept in [P] before the experience value exp reaches the deletion threshold θ_{del}.

In this paper, we propose a new *covering* mechanism to decrease the influence of these classifiers. Specifically, we calculate the *expected action prediction* (EAP) for every action of the classifiers in [M] and compare it with a predefined threshold θ_{EAP}. The predicted payoff vector of a candidate action a_i is calculated as:

$$\boldsymbol{P_{a_i}} = \frac{\sum_{cl.a=a_i \wedge cl \in [\mathrm{M}]} cl.\boldsymbol{p} \cdot cl.F}{\sum_{cl.a=a_i \wedge cl \in [\mathrm{M}]} cl.F} \tag{11}$$

where $cl.\boldsymbol{p}$ and $cl.F$ denote the payoff prediction vector and the fitness value of the classifier cl, respectively. The element $P_{a_i}[o]$ in $\boldsymbol{P_{a_i}}$ indicates the payoff prediction calculated by the action set of a_i after the X-OMQ(λ) agent and the opponent execute action a_i and o, respectively. Combining with the current opponent's model, the EAP for each candidate action in [M] can be represented as:

$$\xi_{a_i} = \sum_{o \in A_2} \sigma(s, o) \cdot P_{a_i}[o] \tag{12}$$

If $\max \xi_{a_i} \leq \theta_{EAP}$, the *covering* mechanism is used and the new classifier is created accordingly. In summary, the *covering* mechanism is evoked when [M] does not cover the predefined minimum actions or the maximum EAP of actions in [M] is less than θ_{EAP}.

4.3. Action selection strategy

Current action selection strategies can be divided into undirected (e.g., Boltzmann distribution and $\epsilon - greedy$) and directed (e.g., Bayesian Q-learning [49]) methods. Besides, optimization algorithms have been incorporated into the action selection strategy, such as the cuckoo action-selection (CAS) method [50], which is not highly sensitive to parameter settings. Compared with directed methods, undirected action selection strategies are easier to be implemented. $\epsilon - greedy$ simply suggests the selection of the best action for exploitation trials and a random action in exploration trials. In our case, Boltzmann distribution seems more reasonable by maintaining a distribution of action selection probability. All candidate actions are taken into account, and the action with larger EAP has a higher probability of being selected. To prevent the numerical issues discussed in [3], we choose the adapted Boltzmann distribution. The probability of selecting a_i in situation s is as follows:

$$\pi(a_i) = \frac{e^{(\xi_{a_i} - \max \xi_{a_i})/T}}{\sum_{a \in A_1} e^{(\xi_a - \max \xi_a)/T}} \tag{13}$$

where T is a temperature parameter to adjust the greediness of the policy. When T grows towards infinite, the policy becomes more random. When T decreases toward zero, the policy becomes more greedy. In experiments, T decreases gradually so that the X-OMQ(λ) agent can fully explore the environment in the early stage of training, and then gradually changes to choose the best action.

Different from the algorithms (e.g., Minimax-Q) that select conservative actions, X-OMQ(λ) selects action considering the opponent's model $\sigma(s, o_i)$. If the opponent never takes the action o in the state s, then the payoff prediction $p[o]$ is ignored by the opponent's model.

4.4. Eligibility trace in X-OMQ(λ)

Eligibility trace is a mechanism for recording historical states and actions during learning. When the agent is rewarded from the environment, values of the most recently visited states are also updated in proportion. In X-OMQ(λ), due to the introduction of eligibility trace, not only the classifiers in the previous time step's action set but also classifiers in [P] that can match the historical information recorded in the eligibility trace are reinforced.

X-OMQ(λ) maintains a trace set Ξ and an eligibility trace e to record the historical information. The trace set Ξ consists of historical state-action tuples (s, a, o). The eligibility trace e records the eligibility values of (s, a, o) tuples in Ξ, which determines how to update the classifiers in Section 4.5.2.

After each learning cycle, the eligibility value of every tuple (s, a, o) recorded in Ξ are decayed as follows:

$$e(s, a, o) = \lambda \cdot \gamma \cdot \ e(s, a, o) \tag{14}$$

where $0 < \gamma < 1$ is the discount factor, $0 < \lambda < 1$ is the trace-decay parameter, which determines the rate at which the trace falls. X-OMQ(λ) only maintains tuples whose eligibility values are significant enough. In other words, a tuple (s, a, o) will be deleted from Ξ if its eligibility value is less than the threshold ($e(s, a, o) < \theta_{et}$).

After the X-OMQ(λ) agent and the opponent execute action a and o respectively in situation s, tuple (s, a, o) is added to Ξ if it is not contained by Ξ. We use the replacing trace technique [51] to update the eligibility trace values after the classifiers are updated:

$$e(s, a_i, o_i) = \begin{cases} 1 & a_i = a, \ o_i = o \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

For X-OMQ(λ), the advantage of using replacing trace is that the eligibility trace e only needs to retain the non-zero value

for each situation s. Thus, the space complexity of e is $O(|S|)$, where $|S|$ is the number of situations in the environment. In the worst case, $|S|$ tuples are recorded in Ξ, which means that for each state, there is a corresponding tuple in e. In contrast, if the accumulating trace [51] is adopted, the space complexity of e will be $O(|S| \cdot |A_1| \cdot |A_2|)$, where $|A_i|, i = 1, 2$ indicates the number of possible actions for player i.

4.5. Evolving the classifiers

Once an action is selected, the X-OMQ(λ) agent forms an action set [A] containing all classifiers in the match set [M] that suggests the selected action. After the selected action is executed, a reward r is received from the environment, which is used to update the classifiers in the previous time step's action set $[A]_{-1}$. Besides, the sets $[A]_{et}$ matched with the eligibility trace is also updated. In this section, we discuss the details of the update mechanism of X-OMQ(λ) and the corresponding eligibility trace technique.

4.5.1. Update classifiers in $[A]_{-1}$

Similar to XCS, in the reinforcement part, the previous time step's action set $[A]_{-1}$ is updated. For each classifier in $[A]_{-1}$, the experience parameter exp is incremented by 1:

$$cl.exp \leftarrow cl.exp + 1 \tag{16}$$

The *target prediction* P denotes the expected total discounted future payoff for $[A]_{-1}$, resulting from taking the selected action a_{-1} in the previous situation s_{-1} and continuing the optimal policy thereafter. Formally, P is the sum of reward r_{-1} and the discounted maximal EAP:

$$P = r_{-1} + \gamma \max_{a \in A_1} \xi_a \tag{17}$$

where $0 < \gamma < 1$ is the discount factor, A_1 is the set of actions available for the X-OMQ(λ) agent, and ξ_a is the EAP for action a. We note that the reward r_{-1} is related to the X-OMQ(λ) agent as well as the opponent thus the prediction corresponding to the opponent action $p[o]$ is updated. The standard Widrow-Hoff delta rule is used to update $p[o]$:

$$cl.p[o] \leftarrow cl.p[o] + \beta_1 (P - cl.p[o]) \tag{18}$$

where $0 < \beta_1 < 1$ is a learning rate.

The prediction error ϵ describes the difference between the target payoff after executing the action and the expected payoff using the current opponent's model. Since the opponent's model is constantly updated during learning, the prediction error is updated accordingly. We use the *expected classifier prediction* (ECP) $cl.\xi$ to denote the expected payoff that the classifier cl receives after the X-OMQ(λ) agent executes the action advocated by cl using the current opponent's model:

$$cl.\xi = \sum_{o \in A_2} \sigma(s_{-1}, o) \cdot cl.p[o] \tag{19}$$

where $\sigma(s_{-1}, o)$ is the opponent' model in situation s_{-1} of the previous time step. Then, the prediction error of each classifier in $[A]_{-1}$ is updated by:

$$cl.\epsilon \leftarrow cl.\epsilon + \beta_2 (|P - cl.\xi| - cl.\epsilon) \tag{20}$$

where $0 < \beta_2 < 1$ is a learning rate.

The action set size as is updated according to the numerosity parameter num of the classifiers in $[A]_{-1}$:

$$cl.as \leftarrow cl.as + \beta_3 \left(\sum_{x \in [A]_{-1}} x.num - cl.as \right) \tag{21}$$

where x is a classifier in $[A]_{-1}$, and $0 < \beta_3 < 1$ is a learning rate.

Similar to XCS, the *moyenne adaptative modifiée* (MAM) technique is also used to update the payoff prediction vector $\boldsymbol{p}$, prediction error ϵ, and action set size as. $\beta_i = 1/cl.exp$ $(i = 1, 2, 3)$ when the experience of a classifier is less than $1/\beta_0$, while at the rest of the time, $\beta_i = \beta_0$ $(i = 1, 2, 3)$, where β_0 is a fixed value. In the early learning phase, MAM technique increases the magnitude of each update and makes X-OMQ(λ) less sensitive to the initial value of these parameters.

The fitness F reflects the accuracy of a classifier, and it is updated in the same way with XCS:

$$cl.F \leftarrow cl.F + \beta_4 \left(\frac{cl.k \cdot cl.num}{\sum_{x \in [A]_{-1}} x.k \cdot x.num} - cl.F \right) \tag{22}$$

where x is the classifier in $[A]_{-1}$, $0 < \beta_4 < 1$ is a learning rate, and $cl.k$ denotes the current absolute accuracy of the classifier:

$$cl.k = \begin{cases} 1 & \text{if } cl.\epsilon < \epsilon_0 \\ \alpha \left(\frac{cl.\epsilon}{\epsilon_0} \right)^{-\nu} & \text{otherwise} \end{cases} \tag{23}$$

where α and ν are real values that further differentiate the accuracy of classifiers. ϵ_0 is the error threshold that indicates the maximal error tolerance.

After the parameters are updated, GA and *subsumption* are executed in $[A]_{-1}$ as stated in Section 3.

4.5.2. Update classifiers in $[A]_{et}$

Eligibility trace value describes the importance of a historical state-action tuple (s, a, o) to a previous situation. For every tuple (s, a, o) in Ξ, a set $[A]_{et}$ is formed by matching the situation s and the action a with the classifiers in the population [P]. We update the classifiers in $[A]_{et}$ based on the error between the *target prediction* (17) and the maximum EAP in the previous time step's match set $[M]_{-1}$. Similar ideas can be found in Q(λ)-learning [52], which updates all state-action pairs in the Q-table by the error between the target Q-value and the maximal Q-value of the current state at the previous time step. The update mechanism of $[A]_{et}$ is shown in Fig. 3.

We note that a set $[A]_{et}$ corresponding to a tuple (s, a, o) is not a real action set, i.e., action a needs not actually performed. However, the classifier parameters of the experience exp, the fitness F, and the action set size as should be updated only when the classifier has been selected in a real action set [A]. Therefore, we only update the payoff prediction vector $\boldsymbol{p}$ and the prediction error ϵ of the classifiers in $[A]_{et}$, while other parameters are not changed. Besides, the classifiers in $[A]_{-1}$ will not be updated again in this process. The update rule is as follows:

$$cl.p[o] \leftarrow cl.p[o] + \beta_5 (P - \Phi_{-1}) \cdot e(s, a, o) \tag{24}$$

$$cl.\epsilon \leftarrow cl.\epsilon + \beta_6 (|P - \Phi_{-1}| - cl.\epsilon) \cdot e(s, a, o) \tag{25}$$

where $0 < \beta_5 < 1$ and $0 < \beta_6 < 1$ are learning rates, $e(s, a, o)$ is the eligibility value for the tuple (s, a, o), and Φ_{-1} is the maximum EAP in $[M]_{-1}$ calculated as follows:

$$\Phi_{-1} = \max_{a_i \in A_1} \sum_{o_i \in A_2} \sigma(s_{-1}, o_i) \cdot P_{a_i}[o_i] \tag{26}$$

where a_i is the action contained in $[M]_{-1}$, $P_{a_i}[o_i]$ is the payoff prediction after the X-OMQ(λ) agent and the opponent execute action a_i and o_i, respectively. In addition, the MAM technique is also used when updating $\boldsymbol{p}$ and ϵ in $[A]_{et}$.

5. Experiments and results

In this section, we will carry out comprehensive evaluations of the X-OMQ(λ) algorithm in the Littman's soccer domain [6], the hunter prey problem [32], and the Hexcer environment [13]. In Section 5.1, the three environments are introduced, and then the experimental setup is described in Section 5.2. Results are given and discussed in Section 5.3.

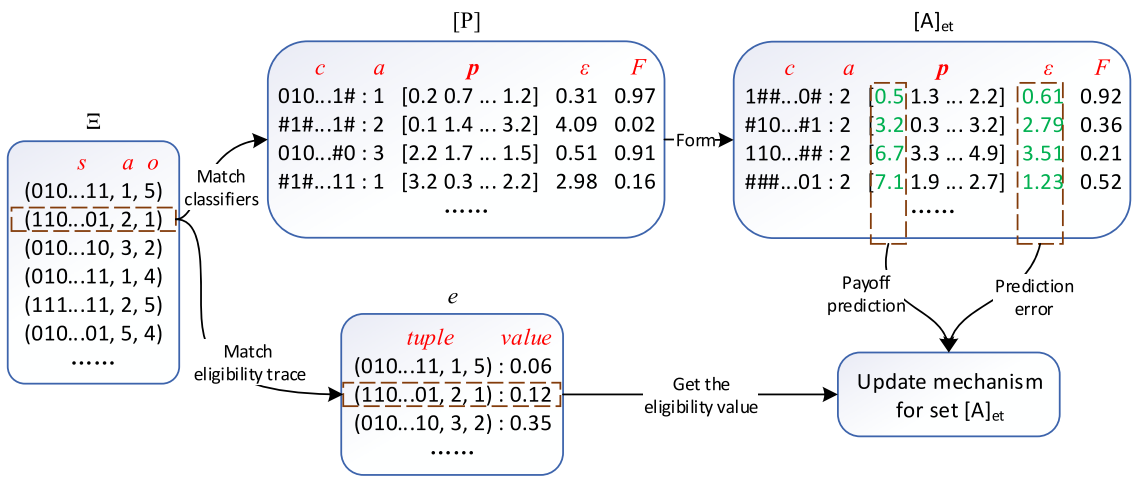


Fig. 3. The workflow of updating $[A]_{et}$. Every tuple (s, a, o) in Ξ is matched with the classifiers in [P] to form $[A]_{et}$. Meanwhile, the corresponding eligibility value for the tuple (s, a, o) is obtained from the eligibility trace e. Then, the error ϵ and the corresponding prediction $p[o]$ of the classifiers in $[A]_{et}$ are updated.

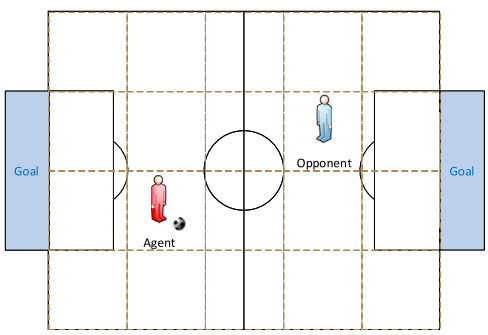


Fig. 4. Littman's soccer domain.

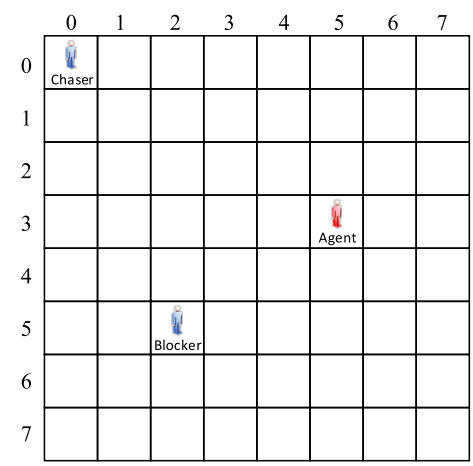


Fig. 5. The hunter prey problem.

5.1. Experimental environment

The Littman' soccer domain is shown in Fig. 4, which is a classical benchmark for ZSMG. The agent and the opponent are placed in a 4 × 5 grid world, the initial states of players are shown in Fig. 4, and the ball is assigned randomly to the agent or the opponent at the start of each game. The player who gets the possession of the ball executes action firstly. Each grid can only be occupied by one player, and the ball occupies the same grid with one of the players. At each time step, a player chooses an action from the action set $A_i = (up, down, left, right, standby), i = 1, 2$.

The player's objective is to bring the ball into the opponent's goal area (as shown in the shadow area in Fig. 4) and score as many points as possible in a match. When a player with the ball moves towards an occupied grid, the player loses the ball and the move fails. Any action that tries to move the player out of the bounds is ignored. The game is ended when any player brings the ball into the opponent's goal area or after a predefined number of time steps has passed. Once a game is ended, the positions of the players and the ball will be reset.

The hunter prey problem is shown in Fig. 5, in which the opponents consist of the chaser and the blocker. The initial state of the agent and the opponents are shown in Fig. 5. At each time step, the agent chooses an action from the action set $A_1 = (up, down, left, right)$. The opponents follow the policy proposed in [53]. In details, the chaser follows the agent movement, while the blocker moves in a horizontal direction to the agent. When the chaser and the blocker collide in the same position, the moves fail. The game is ended when the agent is caught by one of the opponents or a predefined number of time steps has passed. Once the game ends, the positions of the three players are reset.

The Hexcer environment is shown in Fig. 6, which is a more complicated soccer domain. Two players are located in the map made up of connected multiple hexagonal grids. The initial states of the players and the ball are shown in Fig. 6. Unlike the Littman's soccer domain, the initial state of the ball in Hexcer is in the center of the map.

The action set for players in Hexcer is: $A_i = (upper\ left, upper\ right, right, lower\ right, lower\ left, left, standby, i = 1, 2)$. Once a player moves to the grid that has the free ball, then the ball belongs to the player until it is taken by the opponent. When two players collide in the same position while one player is taking the ball, the ball will be randomly reassigned to one of the players and the move fails. If the players try to move out of Hexcer, the move fails and the player remains in its grid. The scoring rules of the game are similar to those in Littman's soccer domain. When a player brings the ball into the opponent's goal, the game ends and the player gets a point. Otherwise, the game is ended after a predefined number of time steps has passed. Once the game is ended, the scenario will be reset.

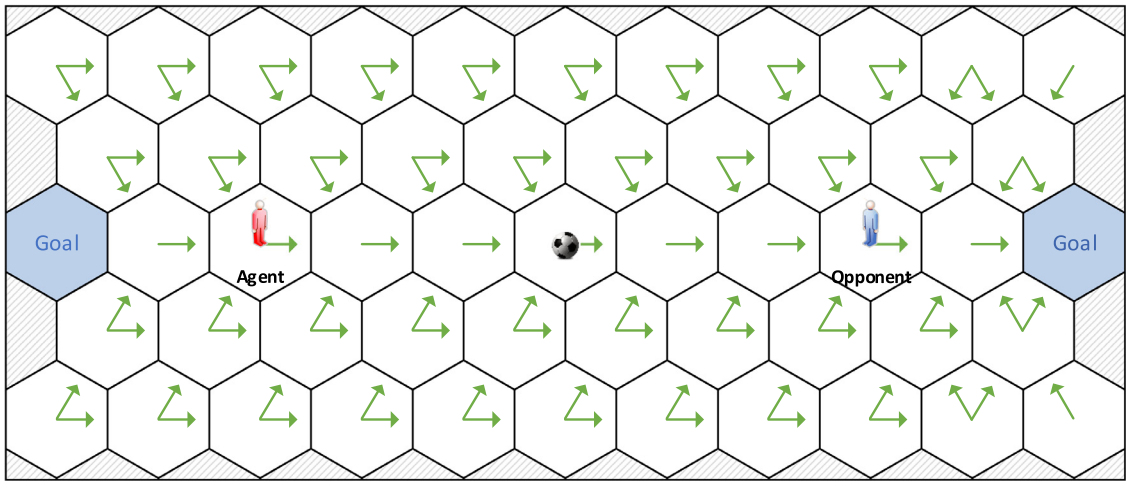


Fig. 6. Hexcer environment and the heuristic policy of the HAMMQ agent used in the experiments. The heuristic policy of the HAMMQ agent is presented as the arrows.

5.2. Experimental setup

We compared the performance of the proposed X-OMQ(λ) algorithm with a variety of QbRL algorithms, including Minimax-Q [6], Minimax-QS [2], Minimax-Q(λ) [8], Minimax-SARSA(λ) [8], NSCP-z [14], and HAMMQ [7], and NNbRL algorithms, including DQN [17], A3C [18], MADDPG [15], PPO [21], and DPPO [20] in the environments introduced in Section 5.1. In the experiments, each player can observe the position and actions of other players. The state of each player consisted of the positions of all players. We use NSCP-z to represent the application of NSCP in zero-sum games. If NSCP [14] is applied to deterministic policy zero-sum games, its best-response solver is nearly identical to the opponent modelling approach in [13]. Moreover, the workflow of the experiments is shown in Fig. 7.

For QbRL algorithms (Minimax-Q, Minimax-QS, Minimax-Q(λ), Minimax-SARSA(λ), NSCP-z, and HAMMQ), the parameters of the above algorithms were almost identical to the original work. In these algorithms, the learning rate α was initialized to 1.0 and decayed at a rate of 0.9999954 for each iteration. The exploration rate was 0.2 and the discount factor $\gamma = 0.9$. Besides, the Q-values of all (s, a, o) tuples and the value function $V(s)$ of all states were initialized to 0. For Minimax-QS, the linearly decreasing spreading function was the same with [2], where $\sigma_t(s, a, o, s_i, a_i, o_i) = g_t(s, s_i) = \tau^d$, $\tau = 0.7$, and $d_{ch} = 1$. For Minimax-Q(λ) and Minimax-SARSA(λ), the trace-decay parameter $\lambda = 0.05$. For HAMMQ, $\xi = \beta = \eta = 1$, which were the same with [7,9].

For X-OMQ(λ), we used the ternary alphabet consisting of $\{0, 1, \#\}$ to represent the condition of the classifiers as with the original XCS. We encoded the grids in the two soccer environments from left to right and from top to bottom, which was consistent with the compared QbRL algorithms. Take Littman's soccer domain as an example, the initial state of players for QbRL algorithms was presented as (11, 8), which was composed by the grid number the player located. Correspondingly, the situation of the initial state for X-OMQ(λ) was presented as 01011 01000, which was constituted by the players' binary coded state. This situation representation may not the best choice because only the position of the players was included, but no other information was used for comparison with other QbRL algorithms above. Parameters initialization of X-OMQ(λ) were shown as Table 5 in Section A.1, most of which were the same with the original XCS.

For NNbRL algorithms (DQN, A3C, MADDPG, PPO, and DPPO), we adopted a similar network structure, each neural network had a hidden layer of 200 units. See Table 6, 7, 8, 9 and 10 in

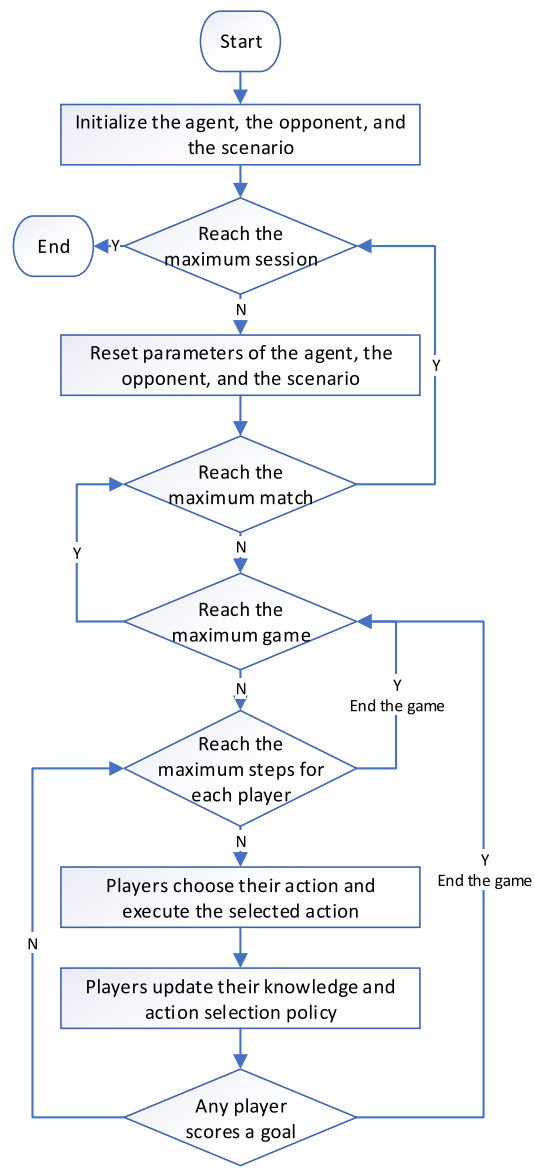


Fig. 7. The workflow of the experiments.

Appendix A for detailed parameters. Moreover, the state representation in these algorithms was consistent with the proposed X-OMQ(λ) in the experiments.

5.2.1. Experiments in Littman's soccer domain

In this environment, we conducted experiments to evaluate the performance of the proposed X-OMQ(λ) compared with the QbRL algorithms, in which the agents were confronted with a random opponent. Once the player scored a goal, it would receive a reward $r = 1$ from the environment, otherwise $r = 0$. The predefined maximum step for every player was 10 and the population size of X-OMQ(λ) was 500. The spreading area used in Minimax-QS and the heuristic policy used in HAMMQ were the same with the original works. A sequence of 100 sessions were run for each learning algorithm and each session consisted of 500 matches of 10 games.

5.2.2. Experiments in hunter prey problem

In the hunter prey problem, we evaluated the performance of X-OMQ(λ) with NNbRL algorithms including A3C, MADDPG, and DPPO. In this case, the agent receive a reward $r = -10$ if it was captured, otherwise $r = 1$. Besides, the chaser and the blocker were considered as a whole opponent. In this way, the action space of the opponent was the combination of the chaser and the blocker. The game was ended if the agent was caught or after each player took 100 steps. In this scenario, the state representation was the same for all algorithms, which was made up of the coordination encoded in binary of all players. For example, the initial state in Fig. 5 was expressed as 011 101 000 000 101 001, which was composed of the coordination of the agent, the chaser, and the blocker. Specifically, every 6 digits represented the coordination of a player, where the first 3 digits represented the row and the last 3 digits were for the column. The population size was set 1000 initially for the X-OMQ(λ) agent. In this experiment, we ran a sequence of 50 sessions for each algorithm and each session consisted of 2000 games.

5.2.3. Experiments in Hexcer environment

In Hexcer, we compared the proposed algorithm with the QbRL and the NNbRL algorithms and carried out more challenging experiments. A random player was also selected as the opponent for the agents. In order to evaluate the performance of the agents against the opponent with learning ability, we selected HAMMQ, which speeding up the learning process through heuristic information, as the opponent. In summary, there were 2 serials of experiments carried out in Hexcer:

1) RL agents versus a random opponent.
2) RL agents versus a HAMMQ opponent.

For each experiment, we ran a sequence of 50 sessions, each of them consisting of 3000 matches. Each match consisted of 10 games, and the maximum step was 50 for each player. A player would obtain a reward of $r = 100$ if it scored one goal, otherwise $r = 0$.

The population set size of the X-OMQ(λ) agent was initialized to 1000. The heuristic policy for HAMMQ agent was shown as the arrows in Fig. 6. Moreover, the heuristic policy for the HAMMQ opponent was symmetric with the HAMMQ agent. As can be seen from Fig. 6, the heuristic policy required a comprehensive and accurate prior knowledge of the environment, in which the heuristic information for each state was targeted.

5.3. Results

We note that Minimax-Q, Minimax-QS, Minimax-Q(λ), and Minimax-SARSA(λ) will be abbreviated as MMQ, MM-QS, MM-Q(λ), and MM-S(λ) in figures of the rest of the paper.

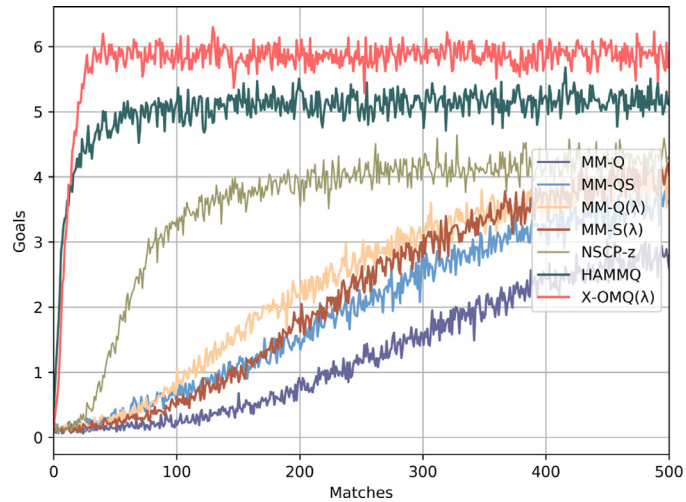


Fig. 8. Averaged goals in each match for the QbRL agents versus a random opponent in the Littman's soccer domain.

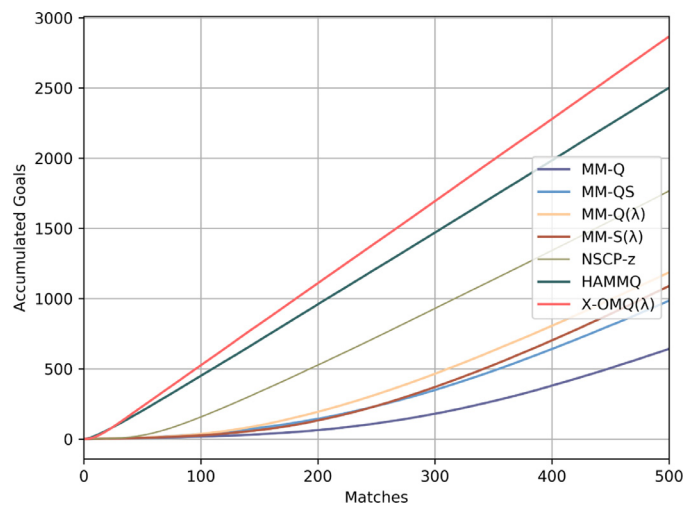


Fig. 9. Accumulated goals for the QbRL agents versus a random opponent in the Littman's soccer domain.

5.3.1. Results in Littman's soccer domain

Figs. 8 and 9 illustrate the averaged and accumulated goals of the RL agents versus the random opponent, respectively. It clearly shows that the X-OMQ(λ) agent gets the best performance. The Minimax-SARSA(λ), the Minimax-Q(λ), and the Minimax-QS agent behave similarly in the experiment. Elicited by the heuristic policy, the HAMMQ agent averages nearly 5 goals at the beginning and remains stable throughout the learning process. While NSCP-z and X-OMQ(λ) maintain the same opponent's model, it is clear that our X-OMQ(λ) agent achieves a better performance leveraging the advantage of XCS and eligibility trace for classifier evolving. In detail, the classifiers that are obviously useless are deleted from the population, which makes the action selection more reasonable and targeted for given states. Besides, once the X-OMQ(λ) agent receives a reward from the environment, the classifiers matched in the trace set Ξ are also updated to improve the learning efficiency.

To further compare the performance of the X-OMQ(λ) agent and the HAMMQ agent versus the random opponent, Fig. 10 presents the T-test results for each match between these two agents. It is evident that after about 25 matches, most of the T values are above the 5% threshold line, which means a significant performance improvement for X-OMQ(λ) compared with HAMMQ.

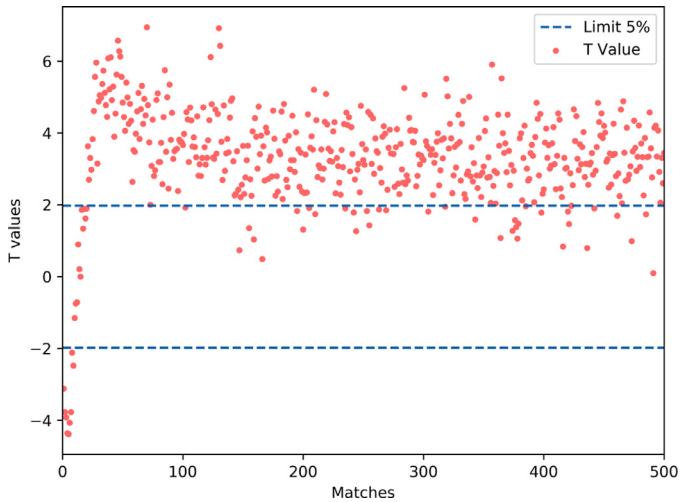


Fig. 10. T-test results for the X-OMQ(λ) agent and the HAMMQ agent versus a random opponent in the Littman's soccer domain.

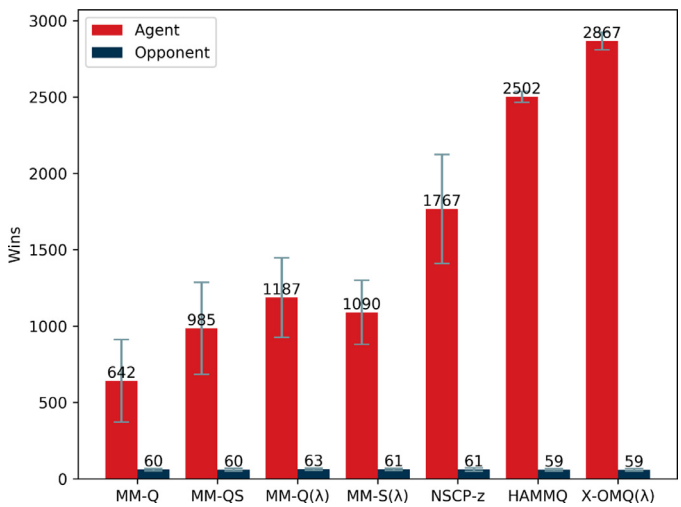


Fig. 11. Averaged cumulative number of wins in 500 matches of the QbRL agents versus a random opponent in the Littman's soccer domain. The gray lines refer to the standard deviation.

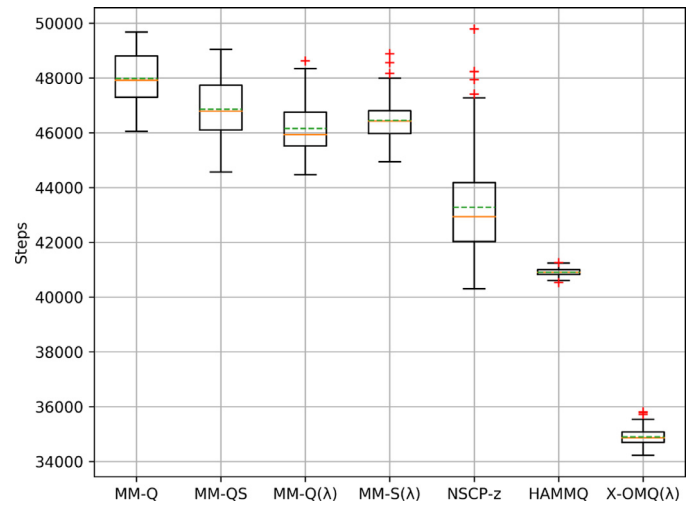


Fig. 12. Averaged total steps in 500 matches of each player in the Littman's soccer domain. The green dotted line refers to the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

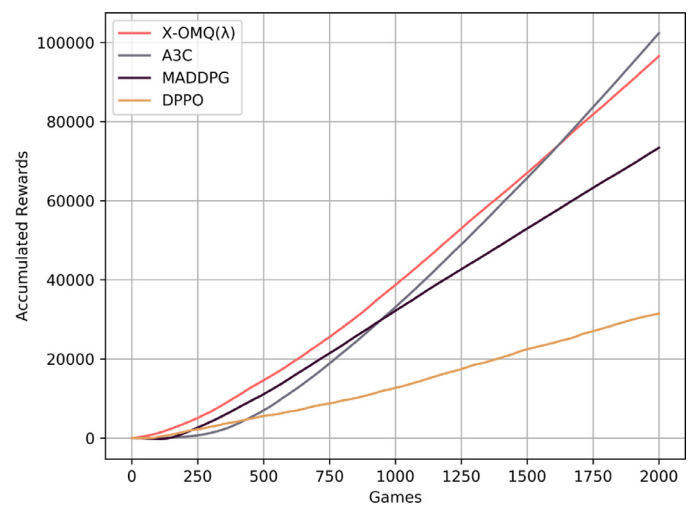


Fig. 13. Accumulated reward for the RL agents in hunter prey problem.

The averaged number of wins in Fig. 11 combing with the averaged total steps in Fig. 12 can be used as a criterion to evaluate the learning performance of the agent. This is because if the game ends in only a few steps, it indicates that one player's policy is dominant. It clearly shows that the X-OMQ(λ) agent obtains the most averaged cumulative number of wins while using the minimum averaged total steps.

From the above results, it is clear that the X-OMQ(λ) agent performs best among the seven RL agents without the need of any prior knowledge. The QbRL algorithms maintain a Q-table containing $20 \times 20 \times 5 \times 5 = 10000$ entities. In contrast, the X-OMQ(λ) agent only maintains an average of 242 *macroclassifiers* in population [P] after learning of each session, among which, only 141 *macroclassifiers* has been selected into action sets (i.e., *cl.exp* > 0). In other words, our proposed algorithm achieves good performance with only a few frequently used classifiers.

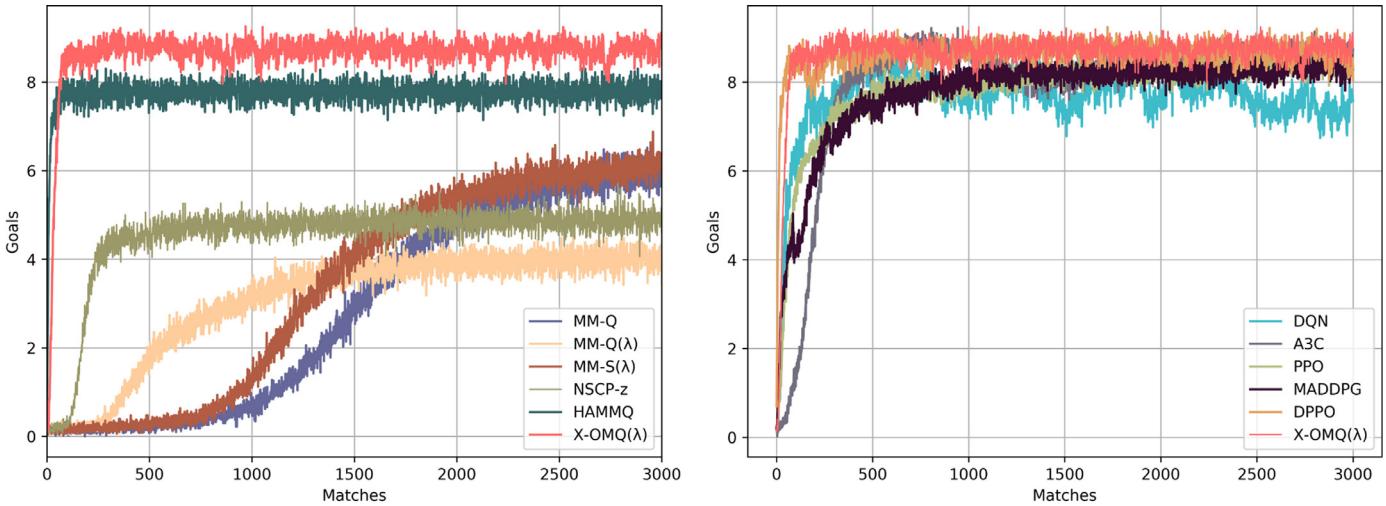
5.3.2. Results in hunter prey problem

In this section, we present the accumulated rewards of the RL agents in the hunter prey problem. Fig. 13 shows the comparison of X-OMQ(λ) with NNbRL algorithms, where the proposed algorithm performs better than MADDPG and DPPO. In details, X-OMQ(λ) accumulates the rewards a little faster than A3C and MADDPG in the first 1600 games, while A3C performs better after that. This is because there are 4 parallel actor-learners updating the shared model together. The results show that X-OMQ(λ) has similar learning performance to the NNbRL algorithms, such as A3C.

In this experiment, X-OMQ(λ) maintains an average of 497 *macroclassifiers* in population [P] after learning, of which 249 have been selected for action selection (i.e., *cl.exp* > 0). The generalization and interpretability of the classifiers are described thoroughly in Section 5.3.5.

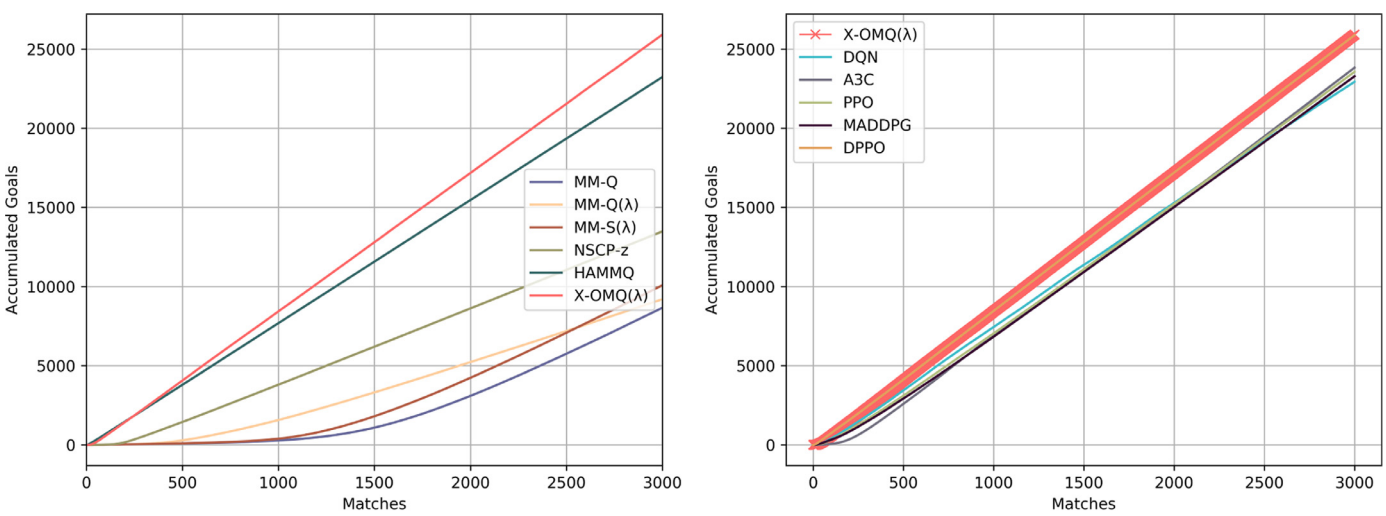
5.3.3. Results of against a random opponent in Hexcer

Fig. 14(a) and Fig. 15(a) present the averaged and accumulated goals of the QbRL agents against a random opponent. The HAMMQ agent scores nearly 8 goals at the start due to the guidance of the heuristic policy in Fig. 6. Using the opponent's model, the NSCP-z agent only performs well in the early stage, while is almost stable after 400 matches. Minimax-Q and Minimax-SARSA(λ) get similar performance in this experiment and outperform NSCP-z after 2000 matches. The X-OMQ(λ) agent shows an excellent learning performance in the early stage without prior knowledge. After about



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 14. Averaged goals in each match for the RL agents versus a random opponent in Hexcer.



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 15. Accumulated goals for the RL agents versus a random opponent in Hexcer.

100 matches, advantages of the HAMMQ agent is outperformed by our method.

Fig. 16(a) and Fig. 17(a) present the averaged cumulative number of wins and the averaged total steps for 3000 matches of each QbRL player, respectively. Due to the guidance of the heuristic policy for each state in Fig. 6, the averaged total steps of the HAMMQ agent in Fig. 17 are the most stable. The X-OMQ(λ) agent gets the most wins in Fig. 16(a) using the least steps without any prior knowledge. In detail, its averaged total steps are nearly 20,000 fewer than the of HAMMQ, while the number of wins is 2500 more than that of HAMMQ. In other words, the X-OMQ(λ) agent has learned a better dominant policy than the HAMMQ agent.

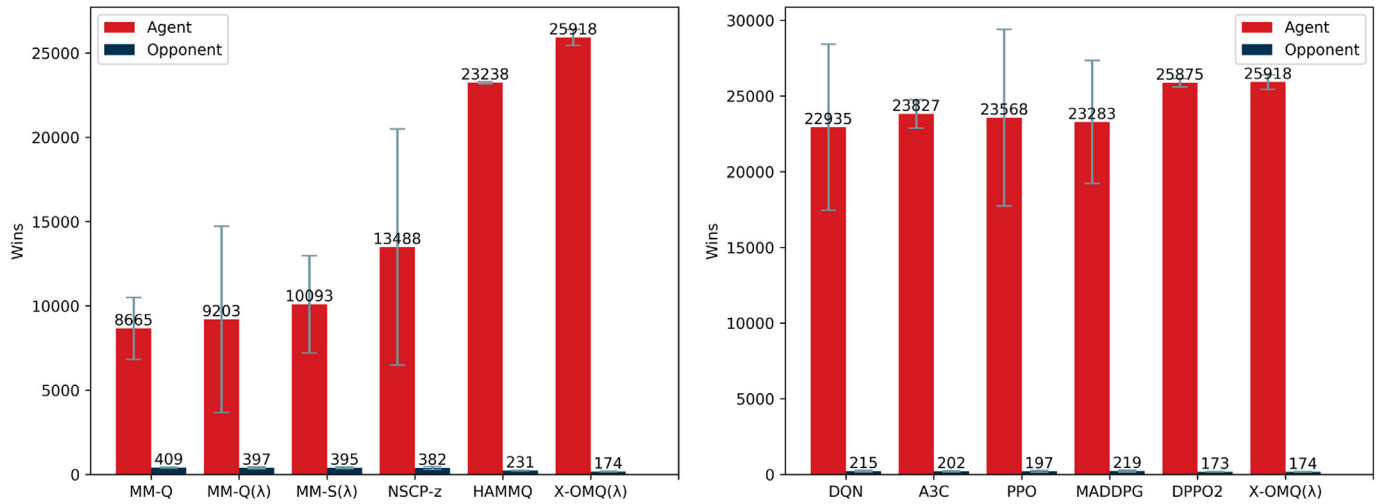
Fig. 14(b) and Fig. 15(b) show that X-OMQ(λ) performs similarly to DPPO during the whole learning process. We note that DPPO presents the best performance among the NNbRL algorithms in this experiment. In details, X-OMQ(λ) and DPPO perform better than other algorithms in the first 500 matches. After 1500 matches, except for DQN, the performance of other algorithms tends to be consistent.

Similar results can be found in Fig. 16(b) and Fig. 17(b). It clearly shows that X-OMQ(λ) and DPPO win the most games with the least number of averaged total steps. The above results present that X-OMQ(λ) performs similar learning performance to NNbRL algorithms. This is due to the fact that our method maintains a fixed-size of the population while evolving the maximally general and accurate classifiers leveraging the advantage of XCS.

5.3.4. Results of against a HAMMQ opponent in Hexcer

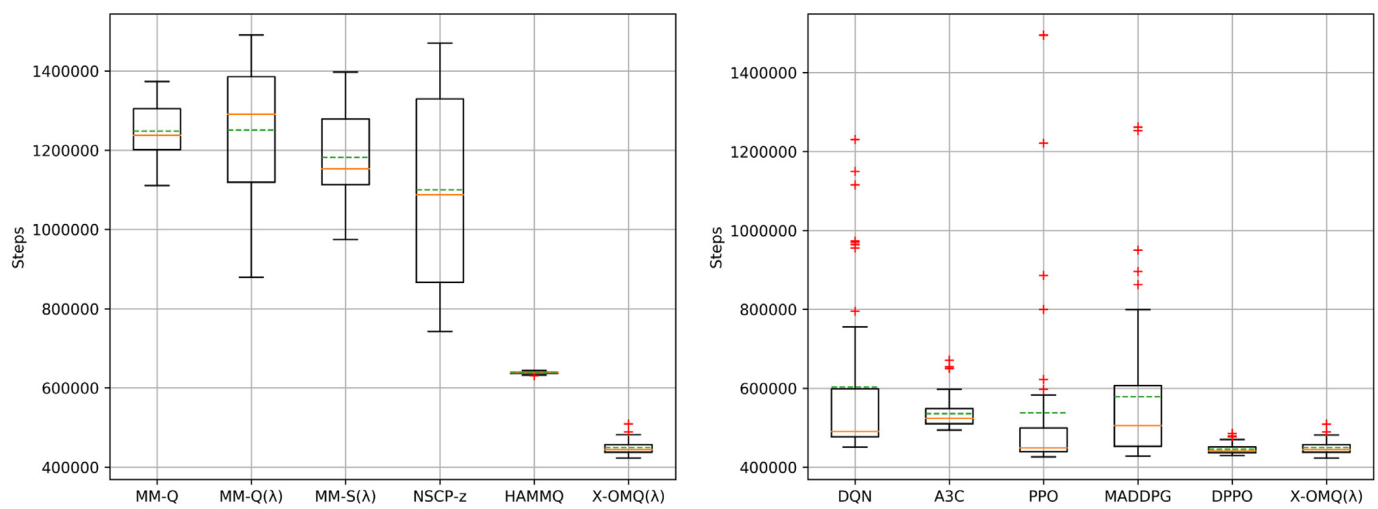
Fig. 18(a) and Fig. 19(a) show the averaged goal balance and accumulated goals of the QbRL agents against the HAMMQ opponent. Minimax-Q, Minimax-Q(λ), Minimax-SARSA(λ), and NSCP-z get similar performance with more than -4 averaged goal balance. The averaged goal balance of HAMMQ fluctuates around 0 during the learning process because they use the same learning algorithm.

In contrast to X-OMQ(λ), OMQ(λ) is its Q-table version without the XCS framework. And the parameters used in OMQ(λ) are identical to X-OMQ(λ) in this experiment. While after about 200 matches in Fig. 18(a), the performance of the OMQ(λ) agent begins to drop blow X-OMQ(λ) clearly, and the averaged goal balance



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 16. Averaged cumulative number of wins in 3000 matches of the RL agents versus a random opponent in Hexcer. The gray lines refer to the standard deviation.



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 17. Averaged total steps in 3000 matches of the RL agents versus a random opponent in Hexcer. The green dotted line refers to the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

is stable around -2. It indicates that the improved XCS and the update mechanism for $[A]_{et}$ play an important role in X-OMQ(λ). The comparison with the QbRL algorithms shows that X-OMQ(λ) demonstrates an excellent learning ability, and after about 400 matches, it outperforms HAMMQ.

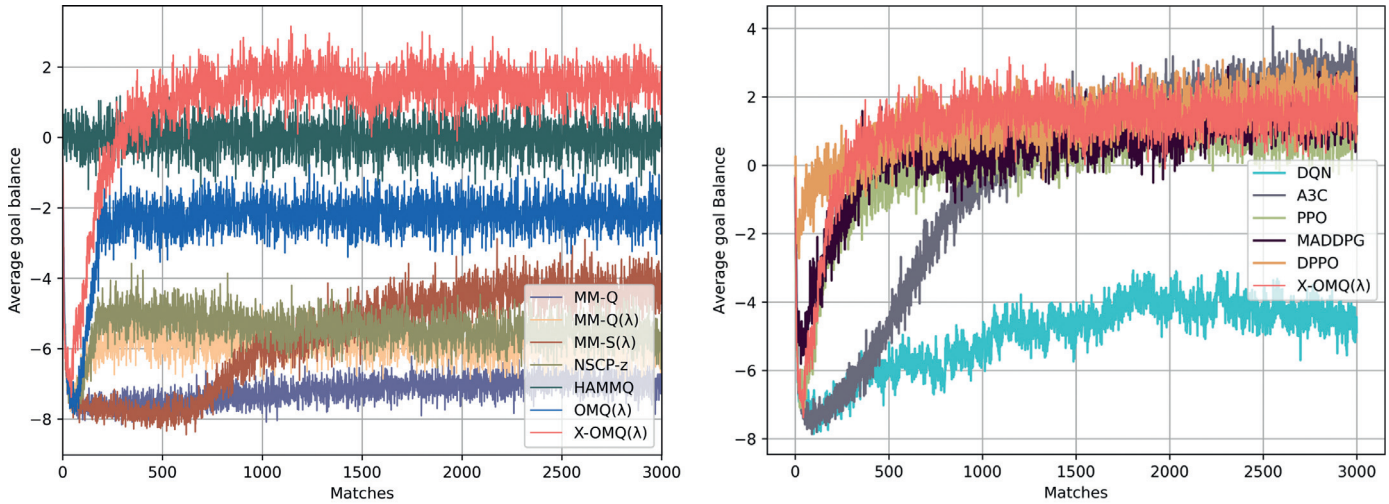
Fig. 20(a) and Fig. 21(a) present the averaged cumulative number of wins and the averaged total steps for 3000 matches of the QbRL agents against a HAMMQ opponent, respectively. When Minimax-Q, Minimax-Q(λ), Minimax-SARSA(λ), and NSCP-z compete against the HAMMQ opponent, the initiative is controlled by the opponent. As a result, their averaged total steps are very close and stable. When HAMMQ confronts itself, both of them use the same learning algorithm. As a result, it performed similarly with the opponent. In contrast, X-OMQ(λ) achieves the best performance while ensuring that the averaged total number of steps is small, and the averaged goal balance reaches about 2, demonstrating the learning performance of our algorithm.

Comparison with the NNbRL algorithm in Fig. 18(b) and Fig. 19(b) show that only DPPO performs slightly better than X-OMQ(λ). Except for DPPO, MADDPG is the most similar algorithm to the performance of X-OMQ(λ). However, our approach obtains more wins than MADDPG and the total number of steps is more evenly distributed as shown in Fig. 20(b) and Fig. 21(b). It indicates that the performance of our approach is more stable. The above results show X-OMQ(λ) behaves a similar performance as the state-of-the-art NNbRL algorithms, such as DPPO. Besides, the learned classifiers are also interpretable, which is detailed in Section 5.3.5.

5.3.5. Generalization and interpretability of classifiers

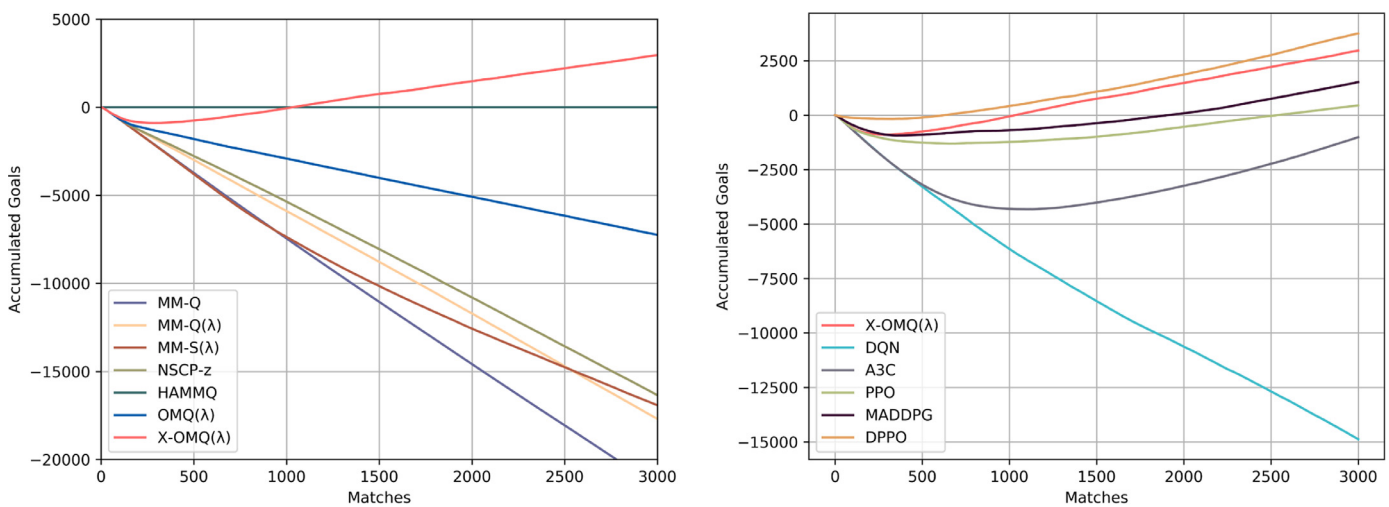
Due to the space limitation, we only present the generality and interpretability of the classifiers learned in Littman's soccer domain and hunter prey problem.

Table 1 gives examples of *macroclassifiers* after learning against a random opponent in Littman's soccer domain. In this environment, the X-OMQ(λ) agent learns the most accurate and general classifiers. Specifically, the errors of the 2nd and the 3rd classifiers



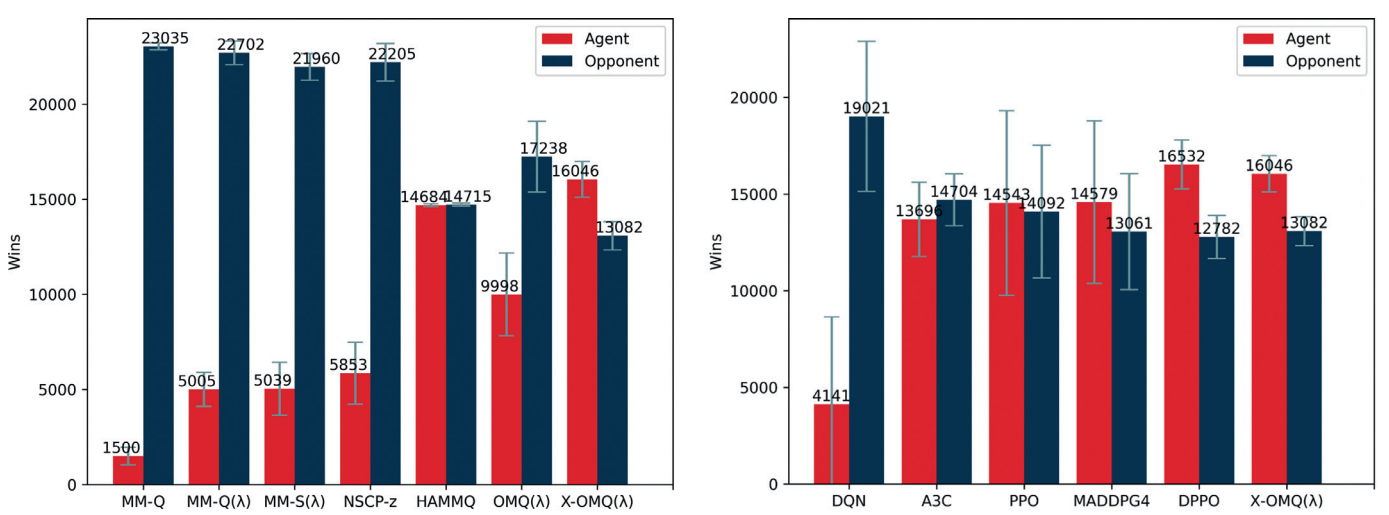
(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 18. Averaged goal balance in each match for the RL agents versus a HAMMQ opponent in Hexcer.



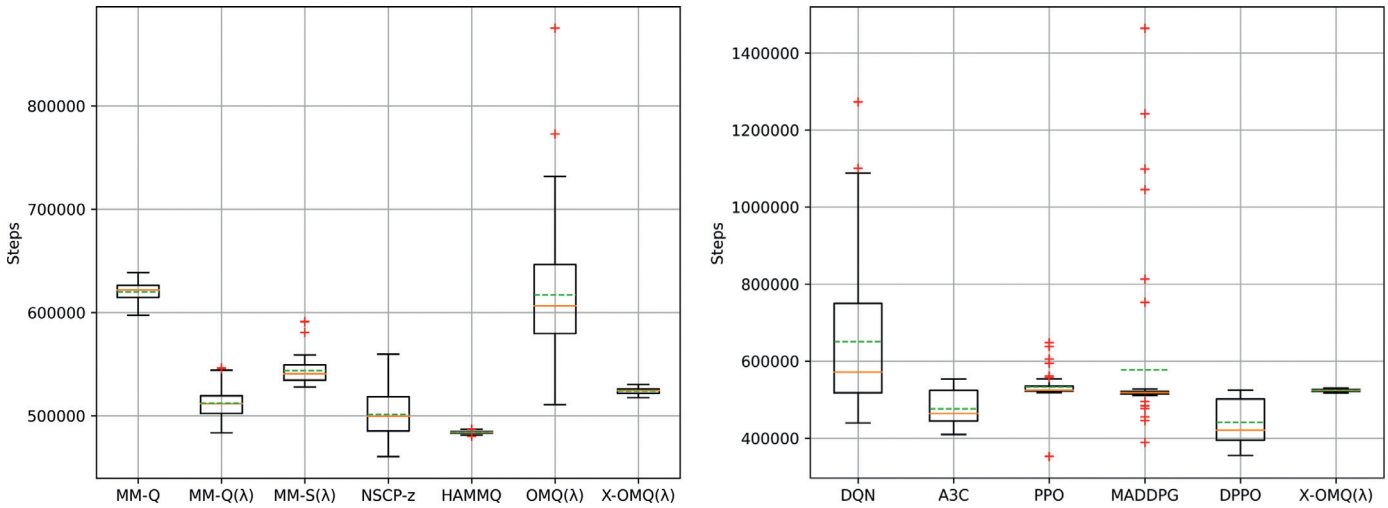
(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 19. Accumulated goal balance in each match for the RL agents versus a HAMMQ opponent in Hexcer.



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 20. Averaged cumulative number of wins in 3000 matches of the RL agents versus a HAMMQ opponent in Hexcer. The gray lines refer to the standard deviation.



(a) Comparison of X-OMQ(λ) with QbRL algorithms. (b) Comparison of X-OMQ(λ) with NNbRL algorithms.

Fig. 21. Averaged total steps in 3000 matches of the RL agents versus a HAMMQ opponent in Hexcer. The green dotted line refers to the mean. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 1
Examples of learned classifiers against a random opponent in the Littman's soccer domain.

No.	Condition	Action	Prediction vector	Fitness	Error	Experience	Numerosity
1	0##101##00	Right	[0.08 0.08 0.08 0.11 0.07]	0.03	0.32	126	2
2	###00#####	Right	[0.08 0.08 0.08 0.08 0.08]	1.00	0.00	4847	90
3	##0#######	Right	[0.08 0.08 0.08 0.08 0.08]	1.00	0.00	4918	158
4	10011####1	Up	[0.12 0.00 0.00 0.00 0.00]	0.28	0.03	21	2

Table 2
Examples of learned classifiers in the hunter prey problem.

No.	Condition	Action	Prediction vector	Fitness	Error	Experience	Numerosity
1	11010#1001##101#01	Down	[0.00 0.00 ··· 3.13 0.00]	0.94	-0.08	17	1
2	###################	Down	[0.00 0.16 ··· 1.13 0.00]	0.97	0.00	56,512	314
3	###################	Up	[3.42 0.05 ··· 3.45 0.00]	0.97	0.00	56,166	312
4	###################	Down	[0.00 0.00 ··· 0.63 0.00]	1.00	0.00	37,048	201

evolve to 0.00 after the experiment, and the wildcards # occupies most of the bits in these classifiers. Besides, because the opponent adopts a random policy i.e., each candidate action in each state has the same probability of being selected, after sufficient iterations, the optimal opponent's model in X-OMQ(λ) has the same probability of performing each action in each statec. As a result, the most accurate and general classifiers (the 2nd and 3rd classifiers) converge to the prediction vector $\boldsymbol{p}$ with identical elements. It indicates that if the agent's situation matches with the condition of the classifiers and the agent selects the action *right*, then it will get the same reward regardless of the opponent action.

We indicate that X-OMQ(λ) has the generalization capability because, in the application phase, the classifier can be used in any situation s that matches the condition c of the classifier. Moreover, we can also obtain a quantitative description of the accuracy of a classifier according to its fitness. In addition, some learned classifiers for the specific states are interpretable. Take the 4th classifier as an example, it can be interpreted as: if the agent is located in the 19th grid (the lower right corner in Fig. 4), the agent should select the action *up*. It indicates that X-OMQ(λ) not only has the generalization ability, but also can learn what human beings can understand.

Table 2 shows several learned *macroclassifiers* in the hunter prey problem. As described in Section 5.2.2, the state for players

Table 3
The number of *macroclassifiers* after learning in Hexcer.

Opponent	Number	Selected
Random	773	262
HAMMQ	720	267

was constituted with the binary-coded coordination of the agent, the chaser, and the blocker, respectively. As a result, the 1st classifier in Table 2 can be interpreted as: if the agent is located in the sixth row and the fourth or fifth column, the chaser is in the fourth row and its column number is greater than three, and the blocker is in the fifth row and the first or fifth column, the agent should choose action *down*. This classifier is reasonable because in most of these situations, the two opponents collide with each other according to their policy and thus the move fails. Besides, we also obtain the most general and accurate rules, such as the 2nd, 3rd, and 4th classifiers, in which the fitness is close to 1.0 and the error is 0.0.

Population size in *macroclassifiers* is a measure of the space complexity of the XCS [23]. In the two experiments in Hexcer, the number of *macroclassifiers* averaged over 50 sessions is shown in Table 3, and the number of selected *macroclassifiers* in the action

Table 4
Comparison of learning time in each session against a HAMMQ opponent in Hexcer.

Algorithms	Learning Time (s) ($\pm$ std)
Minimax-Q	839.7 ($\pm$ 139.5)
Minimax-Q(λ)	764.3 ($\pm$ 37.5)
Minimax-SARSA(λ)	1716.4 ($\pm$ 59.6)
NSCP-z	753.8 ($\pm$ 44.9)
HAMMQ	1928.7 ($\pm$ 1360.9)
OMQ(λ)	1068.8 ($\pm$ 139.6)
DQN	1035.5 ($\pm$ 325.5)
A3C	1081.1 ($\pm$ 156.3)
A3C(GPU)	740.6 ($\pm$ 64.7)
PPO	886.7 ($\pm$ 45.3)
PPO(GPU)	792.8 ($\pm$ 67.9)
MADDPG	540.5 ($\pm$ 252.2)
DPPO	1011.2 ($\pm$ 224.1)
X-OMQ(λ)	2549.2 ($\pm$ 119.2)

sets ($cl.exp > 0$) is also listed. We note that X-OMQ(λ) achieves similar performance to the state-of-the-art RL algorithms only using a relatively small number of classifiers.

5.3.6. Learning time analysis

In this section, we compare the time needed in each session of the proposed algorithm with RL algorithms against a HAMMQ opponent in Hexcer. The algorithms were implemented in Python, and the experiments performed in a Microsoft Windows computer with 16-Core Intel i9-9900K CPU, 64GB of RAM and GeForce GTX 2080Ti GPU. Besides, the neural networks were implemented in TensorFlow 1.14.0. The summarized results are averaged over 50 sessions and shown in Table 4.

It shows that X-OMQ(λ) requires much more time in the learning phase compared with algorithms with similar performance, such as A3C and DPPO. This is because GA and the eligibility trace mechanism consume most of the time in the classifiers update process.

Summarizing the above results, X-OMQ(λ) shows an excellent learning performance and outperforms the QbRL algorithms using a relatively small number of classifiers. Its performance exceeds the algorithms with prior knowledge, such as HAMMQ. Moreover, X-OMQ(λ) behaves similar learning performance to the state-of-the-art NNbRL algorithms, such as DPPO and A3C. Furthermore, the learned classifiers are also interpretable while having generalization capabilities. One deficiency of X-OMQ(λ) is that it requires more time in the learning phase compared with the NNbRL algorithms.

6. Conclusion and future works

In this paper, we have proposed the X-OMQ(λ) algorithm to learn general and interpretable action selection policies in ZSMG. The opponent's model runs through the entire process of agent modelling, action selection, and learning. Elicited by the opponent's model, the X-OMQ(λ) agent chooses action in a best-response manner. Besides, the incorporation of the improved XCS generalizes the state-action space and obtains interpretable action selection rules. Moreover, the eligibility trace mechanism is introduced to the learning part, and the classifiers that can match the historical information are reinforced in $[A]_{et}$.

The comprehensive experiment results have shown that X-OMQ(λ) has similar learning performance with state-of-the-art RL algorithms in competitive Markov Games. One advantage of X-OMQ(λ) is that the learned classifiers are both general and accurate. Another advantage is that X-OMQ(λ) can learn interpretable classifiers. Thus, the explainability of the learning results is better compared with the NNbRL algorithms. Furthermore, the learned interpretable classifiers can be directly used as heuristic rules for other similar but more complex scenarios to speed up learning. Nevertheless, the deficiency of X-OMQ(λ) is that it requires more time in the learning phase due to the process of rule evolution conducted by GA and the eligibility trace mechanism. In addition, compared with the NNbRL algorithms, our method does not yet support images as input and is not suitable for continuous action space. Future work could be adapting the proposed X-OMQ(λ) algorithm to solve more complex tasks in which multiple cooperative agents learn against multiple opponents that are learning concurrently in continuous environment. Besides, we will further optimize the efficiency of the algorithm by, for example, combing efficient action selection methods such as CAS.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Hao Chen: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing - original draft. **Chang Wang:** Conceptualization, Supervision. **Jian Huang:** Conceptualization, Supervision. **Jiangtao Kong:** Formal analysis. **Hanqiang Deng:** Writing - review & editing.

Acknowledgment

This work is supported by the National Nature Science Foundation of China under Grant No.61906203 and the 13th Five-Year equipment pre-research sharing technology project No. 41412030301.

Appendix A. Parameters of the algorithms used in the experiments

A1. Initial parameters of X-OMQ(λ) in the experiments

Table 5
Initial parameters of X-OMQ(λ) in the experiments.

Parameter	Notation	value
Trace decay	λ	0.05
Learning rate	β_i $(i = 1, 2, 3, 4, 5, 6)$	0.15
Accuracy coefficient	α	0.1
Error threshold	ϵ_0	0.01
Accuracy power	ν	5
Discount factor	γ	0.71
GA threshold	θ_{GA}	35
Eligibility trace threshold	θ_{et}	0.001
Covering threshold	θ_{EAP}	0
Cross probability	χ	0.75
Mutation probability	μ	0.03
Deletion threshold	θ_{del}	20
Fitness threshod	δ	0.1
Subsumption threshold	θ_{sub}	20
Wildcard probability	$P_{\#}$	0.33
Initial prediction vector	$\boldsymbol{p}$	[0.00001 0.00001 $\cdots$]
Initial prediction error	ϵ	0.00001
Initial fitness	F	0.00001

A2. Parameters of A3C in the experiments

In the hunter prey problem, the A3C agent was updated every 5 steps, or when it was caught by the opponent. In the Hexcer scenario, the A3C agent was updated every 5 matches, or when the agent scored a goal.

Table 6
Parameters of A3C in the experiments.

Parallel learners	4	Minibatch size	32
Actor learning rate	0.0001	Critic learning rate	0.0001
Entropy parameter	0.001	Optimizer	RMSPropOptimizer
Activation function	ReLU	Discount factor	0.9

A3. Parameters of MADDPG in the experiments

In MADDPG, the estimates player model has a hidden layer of 100 units, and the output layer is a softmax layer with each unit corresponding to the candidate action selection probability of the estimated player. In the hunter prey problem, the MADDPG agent was updated every step. In the Hexcer scenario, the MADDPG agent was updated every 50 steps.

Table 7
Parameters of MADDPG in the experiments.

Memory size	1000	Minibatch size	100
Actor learning rate	0.0001	Critic learning rate	0.0005 (hunter prey) / 0.005 (Hexcer)
Discount factor	0.9	Optimizer	AdamOptimizer
Activation function	ReLU	Soft replacement parameter	0.01
Learning rate for the estimated player model	0.0001	Entropy parameter	0.001

A4. Parameters of DPPO in the experiments

In the hunter prey problem, the DPPO agent was updated every 5 steps, or when it was caught by the opponent. In the Hexcer scenario, the DPPO agent was updated every 5 matches, or when the agent scored a goal.

Table 8
Parameters of DPPO in the experiments.

Parallel workers	4	Minibatch size	32
Actor learning rate	0.001 (hunter prey) / 0.0001 (Hexcer)	Critic learning rate	0.01 (hunter prey) / 0.0001 (Hexcer)
Clipping parameter	0.2	Optimizer	AdamOptimizer
Activation function	ReLU	Soft replacement parameter	0.01
Actor update rounds	10	Critic update rounds	10
Discount factor	0.9		

A5. Parameters of DQN in Hexcer environment

In the Hexcer scenario, the DQN agent began to updated after 500 transactions stored in the memory.

Table 9
Parameters of DQN in Hexcer environment.

Learning rate	0.01	Minibatch size	32
Optimizer	RMSPropOptimizer	Discount factor	0.9
Memory size	1000	Target network	Update every 10 matches
ϵ-greedy	1.0(initial) $\rightarrow$ 0.001		

A6. Parameters of PPO in Hexcer environment

In the Hexcer scenario, the PPO agent was updated every 5 matches, or when the agent scored a goal.

Table 10
Parameters of PPO in Hexcer experiments.

Discount factor	0.9	Minibatch size	32
Actor learning rate	0.0001	Critic learning rate	0.0001
Clipping parameter	0.2	Optimizer	AdamOptimizer
Activation function	ReLU	Soft replacement parameter	0.01
Actor update rounds	10	Critic update rounds	10

References

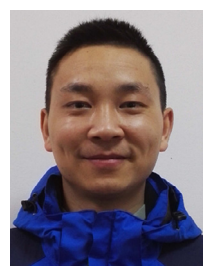
[1] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[2] C.H.C. Ribeiro, R. Pegoraro, A.H.R. Costa, Experience generalization for concurrent reinforcement learners: the Minimax-QS algorithm, in: International Joint Conference on Autonomous Agents & Multiagent Systems, 2002.
[3] H.P. van Hasselt, Insights in Reinforcement Learning, Hado van Hasselt, 2011.
[4] L. Buşoniu, R. Babuška, B. De Schutter, Multi-agent reinforcement learning: an overview, in: Innovations in Multi-agent Systems and Applications-1, Springer, 2010, pp. 183–221.
[5] M. TAN, Multi-agent reinforcement learning:independent vs.cooperative agent, The 10th International Conference on Machine Learning (ICML-93) (1993) 330–337. https://ci.nii.ac.jp/naid/10004577507/en/.
[6] M.L. Littman, Markov games as a framework for multi-agent reinforcement learning, in: Machine Learning Proceedings 1994, Elsevier, 1994, pp. 157–163.
[7] R.A. Bianchi, M.F. Martins, C.H. Ribeiro, A.H. Costa, Heuristically-accelerated multiagent reinforcement learning, IEEE Trans. Cybern. 44 (2) (2014) 252.
[8] B. Banerjee, S. Sen, P. Jing, Fast concurrent reinforcement learners, in: International Joint Conference on Artificial Intelligence, 2001.
[9] R.A. Bianchi, C.H. Ribeiro, A.H.R. Costa, Heuristic selection of actions in multi-agent reinforcement learning., in: IJCAI, 2007, pp. 690–695.
[10] C. Ribeiro, C. Szepesvári, Q-learning combined with spreading: convergence and results, in: Procs. of the ISRF-IEE International Conf. on Intelligent and Cognitive Systems (Neural Networks Symposium), 1996, pp. 32–36.
[11] D. Carmel, S. Markovitch, Opponent modeling in multi-agent systems, in: International Joint Conference on Artificial Intelligence, Springer, 1995, pp. 40–52.
[12] G. Tesauro, Extending Q-learning to general adaptive multi-agent systems, in: Advances in Neural Information Processing Systems, 2004, pp. 871–878.
[13] W. Uther, M. Veloso, Adversarial reinforcement learning, Technical Report, Technical report, Carnegie Mellon University, 1997. Unpublished, 1997.
[14] M. Weinberg, J.S. Rosenschein, Best-response multiagent learning in non-stationary environments, in: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2, IEEE Computer Society, 2004, pp. 506–513.
[15] R. Lowe, Y. Wu, A. Tamar, J. Harb, O.P. Abbeel, I. Mordatch, Multi-agent actor–critic for mixed cooperative-competitive environments, in: Advances in Neural Information Processing Systems, 2017, pp. 6379–6390.
[16] Y. Zheng, Z. Meng, J. Hao, Z. Zhang, T. Yang, C. Fan, A deep Bayesian policy reuse approach against non-stationary agents, in: Advances in Neural Information Processing Systems, 2018, pp. 954–964.
[17] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.
[18] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: International Conference on Machine Learning, 2016, pp. 1928–1937.
[19] T. Lillicrap, J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, Continuous control with deep reinforcement learning(2016).
[20] N. Heess, T.B. Dhruva, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S.M.A. Eslami, et al., Emergence of locomotion behaviours in rich environments, arXiv: Artificial Intelligence(2017).
[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, 2017 arXiv:1707.06347.
[22] S.W. Wilson, Classifier fitness based on accuracy, Evol. Comput. 3 (2) (1995) 149–175.
[23] S.W. Wilson, S. Wilson, G. Xcs, et al., Generalization in the XCS classifier system(1998).
[24] J.H. Holland, Adaptation in Natural and Artificial System, 1992.
[25] J.H. Holland, J.S. Reitman, Cognitive systems based on adaptive algorithms, in: Pattern-Directed Inference Systems, Elsevier, 1978, pp. 313–329.
[26] P.L. Lanzi, Learning classifier systems from a reinforcement learning perspective, Soft Comput. 6 (3–4) (2002) 162–170.
[27] J.-B. Li, A.-P. Chen, Refined group learning based on XCS and neural network in intelligent financial decision support system, in: Sixth International Conference on Intelligent Systems Design and Applications, 2, IEEE, 2006, pp. 925–930.
[28] G. Énée, C. Escazut, A minimal model of communication for a multi-agent classifier system, in: International Workshop on Learning Classifier Systems, Springer, 2001, pp. 32–42.
[29] K.-Y. Chen, P.A. Lindsay, P.J. Robinson, H.A. Abbass, A hierarchical conflict resolution method for multi-agent path planning, in: 2009 IEEE Congress on Evolutionary Computation, IEEE, 2009, pp. 1169–1176.

[30] H. Inoue, K. Takadama, K. Shimohara, Exploring XCS in multiagent environments, in: Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation, ACM, 2005, pp. 109–111.
[31] C. Lode, U. Richter, H. Schmeck, Adaption of XCS to multi-learner predator/prey scenarios, in: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, ACM, 2010, pp. 1015–1022.
[32] B.H. Abed-Alguni, S.K. Chalup, F.A. Henskens, D.J. Paul, A multi-agent cooperative reinforcement learning model using a hierarchy of consultants, tutors and workers, Viet. J. Comput. Sci. 2(4) 213–226.
[33] C.J. Watkins, P. Dayan, Q-learning, Mach. Learn. 8 (3–4) (1992) 279–292.
[34] C. Szepesvári, M.L. Littman, A unified analysis of value-function-based reinforcement-learning algorithms, Neural Comput. 11 (8) (1999) 2017–2060.
[35] M. Bowling, M. Veloso, Multiagent learning using a variable learning rate, Artif. Intell. 136 (2) (2002) 215–250.
[36] M. Bowling, Convergence and no-regret in multiagent learning, in: Advances in Neural Information Processing Systems, 2005, pp. 209–216.
[37] B. Banerjee, J. Peng, Adaptive policy gradient in multiagent learning, in: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, ACM, 2003, pp. 686–692.
[38] C. Claus, C. Boutilier, The dynamics of reinforcement learning in cooperative multiagent systems, AAAI/IAAI 1998 (1998) 746–752.
[39] V. Conitzer, T. Sandholm, Awesome: a general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents, Mach. Learn. 67 (1–2) (2007) 23–43.
[40] R. Raileanu, E. Denton, A. Szlam, R. Fergus, Modeling others using oneself in multi-agent reinforcement learning, in: J. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, Proceedings of Machine Learning Research, 80, PMLR, Stockholmsmssan, Stockholm Sweden, 2018, pp. 4257–4266.
[41] H. He, J. Boyd-Graber, K. Kwok, H. Daumé III, Opponent modeling in deep reinforcement learning, in: International Conference on Machine Learning, 2016, pp. 1804–1813.
[42] L.M. Hercog, Better manufacturing process organization using multi-agent self--organization and co-evolutionary classifier systems: the multibar problem, Appl. Soft. Comput. 13 (3) (2013) 1407–1418.
[43] L.M. Hercog, T.C. Fogarty, Social simulation using a multi-agent model based on classifier systems: The emergence of vacillating behaviour in the El Farol bar problem, in: International Workshop on Learning Classifier Systems, Springer, 2001, pp. 88–111.
[44] Z. Guessoum, L. Rejeb, O. Sigaud, Using XCS to build adaptive agents, in: AISB 2004 Convention, 2004, p. 101.
[45] F. Klügl, R. Hatko, M.V. Butz, Agent learning instead of behavior implementation for simulations–a case study using classifier systems, in: German Conference on Multiagent System Technologies, Springer, 2008, pp. 111–122.
[46] S.Y.B. Wong, S. Schulenburg, Portfolio allocation using XCS experts in technical analysis, market conditions, and options market, in: Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, 2007, pp. 2965–2972.
[47] A.J. Bagnall, G.D. Smith, A multiagent model of the UK market in electricity generation, IEEE Trans. Evol. Comput. 9 (5) (2005) 522–536.
[48] U. Richter, H. Prothmann, H. Schmeck, Improving XCS performance by distribution, in: Asia-Pacific Conference on Simulated Evolution and Learning, Springer, 2008, pp. 111–120.
[49] R. Dearden, N. Friedman, S. Russell, Bayesian Q-learning (1998) 761–768.
[50] B.H. Abedalguni, Action-selection method for reinforcement learning based on cuckoo search algorithm, Arab. J. Sci. Eng. 43 (12) (2018) 6771–6785.
[51] S.P. Singh, R.S. Sutton, Reinforcement learning with replacing eligibility traces, Mach. Learn. 22 (1–3) (1996) 123–158.
[52] P. Jing, R.J. Williams, Incremental multi-Step Q-Learning, 1996.
[53] C.H. Yong, R. Miikkulainen, Coevolution of role-based cooperation in multiagent systems, IEEE Trans. Auton. Ment. Dev. 1 (3) (2009) 170–186.

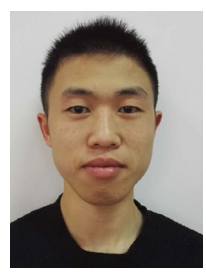
Hao Chen received the Bachelor's degree in electrical engineering and automation from the College of Electrical Engineering, Hebei University of Technology (HEBUT), Tianjin, China, in 2015, and the Master's degree in control science and engineering from the College of Mechatronics and Automation, National University of Defense Technology (NUDT), Hunan, China, in 2017. He is currently a Ph.D. student at the College of Intelligence Science and Technology, NUDT, China. His current research interests include reinforcement learning and multi-agent system.

Chang Wang is an assistant professor at the Institute of Unmanned Systems, National University of Defense Technology, China. He holds his B.S. degree in mathematics from University of Science and Technology of China (USTC), M.Sc. degree in applied mathematics from National University of Defense Technology, and Ph.D. degree in robot learning from Delft University of Technology, the Netherlands. His research interests include reinforcement learning, developmental robotics, and human-robot teamwork.

Jian Huang received the B.S. degree in control science and engineering from the Department of Automatic Control, NUDT, Hunan, in 1994, where she received the Master's degree and Ph.D. degree in control science and engineering from the College of Mechatronics and Automation, NUDT, in 1997 and 2000, respectively. She now is a professor at the College of Intelligence Science and Technology, NUDT. She has been a visiting professor at Technische Universiteit Delft. She is the co-author of the book: HLA simulation system integrated design (Changsha, Hunan, National University of Defense Technology Press, 2008). Her current research interests include system simulation and artificial intelligence.



Jiangtao Kong is a doctoral student with the College of Intelligence Science and Technology, National University of Defense Technology, Hunan, China. His major is task planning and simulation. His current research interests include system simulation and artificial intelligence.



Hanqiang Deng received the Bachelor's degree in Mechanical Engineering and Automation from the School of mechanical engineering and automation, Beijing University of Aeronautics and Astronautics, Beijing, China, in 2016. Now, he is pursuing the Master's degree in control science and engineering with the College of Intelligence Science and Technology, National University of Defense Technology, Hunan, China. His current research interests include robotics and simulation.