Scalable Multi-agent Reinforcement Learning Architecture for Semi-MDP Real-Time Strategy Games

Zhentao Wang(✉), Weiwei Wu(✉), and Ziyao Huang

School of Cyber Science and Engineering, Southeast University, Nanjing, China
jsyxl994@hotmail.com, weiweiwu@seu.edu.cn,
seuhuangziyao@outlook.com

Abstract. Action-value has been widely used in multi-agent reinforcement learning. However, action-value is hard to be adapted to scenarios such as real-time strategy games where the number of agents can vary from time to time. In this paper, we explore approaches of avoiding the action-value in systems in order to make multi-agent architectures more scalable. We present a general architecture for real-time strategy games and design the global reward function which can fit into it. In addition, in our architecture, we also propose the algorithm without human knowledge which can work for Semi Markov Decision Processes where rewards cannot be received until actions last for a while. To evaluate the performance of our approach, experiments with respect to micromanagement are carried out on a simplified real-time strategy game called MicroRTS. The result shows that the trained artificial intelligence is highly competitive against strong baseline robots.

Keywords: Markov Decision Process · Multi-agent systems · Reinforcement learning · Real-Time strategy games · Deep learning · Game theory

1 Introduction

Reinforcement learning (RL), which is a subfield of machine learning, is about learning what to do by try and error so as to maximize a numerical reward signal [1]. Reinforcement learning along with deep neural network has achieved great success universally these years, especially in game areas [2–4]. Encouraged by this, researchers have been further studying on multi-agent reinforcement learning (MARL) algorithms and use games for performance tests. However, solving multi-agent real-time planning problems such as real-time strategy (RTS) games is of great challenge by MARL approaches. The reason is that, unlike single agent games like Atari [5], in RTS games: 1) the number of units one player can control can be quite large and changeable, which indicates that with the increasing units, the state-action spaces grow exponentially; 2) The game is real-time, which means time is limited (usually within a semi-second) for players to make decisions to deploy units at every timestep. 3) Units in RTS games are generally heterogenetic; 4) There exists both competition and cooperation for units.

H. Zhang et al. (Eds.): NCAA 2020, CCIS 1265, pp. 433–446, 2020.
https://doi.org/10.1007/978-981-15-7670-6_36

The four features mentioned above lead to the difficulty of training an efficient RL artificial intelligence (AI) for RTS games.

Existing algorithms designed for RTS games are mostly script-based or search-based. Some of these algorithms typically apply domain knowledge to RTS AI [6–9], the others search the inner game model to look ahead and use a heuristic function to evaluate the simulating results [10, 11]. These AIs usually work well, however, to some extent, the performance depends on how "smart" their designers are, not how smart themselves are. On the contrary, several MARL approaches take little domain knowledge from human, but strong prerequisite such as homogeneous of units or fixed number of units are demanded for the architecture and training algorithms [12].

Thus, it calls for a more flexible multi-agent reinforcement learning architecture to be able to scale to complex scenarios. In this work, we mainly focus our research on a simplified RTS game environment called MicroRTS (we also make it gym-like [13] for RL training). We present a multi-agent reinforcement learning architecture that shares one global critic and multiple switchable actors, which is scalable and general for RTS games. We also give our Semi Markov Decision Process (SMDP) algorithm for this architecture. Then, we train our AI in a self-play manner in MicroRTS environment. Finally, to measure the performance, the trained AI are compared with the traditional script-based and search-based baseline algorithms by combating with each other. Result shows that our self-played AI, as the first AI for MicroRTS game without any human knowledge, outperforms these baseline AIs in micromanagement scenarios.

2 Related Work

In this section, we mainly review relevant works on search-based and MARL methods and respectively take these two kinds into consideration.

2.1 Search-Based Methods

Many AI methods in RTS games are hard-coded by game experts, which may fail to work because other players can easily take advantages of these rule-based methods. Researches on RTS games start from search-based algorithms. NaiveMCTS makes effort to solve the Combinatorial Multi-Armed Bandit (CMAB) problems in RTS games by two Monte Carlo Tree Search (MCTS) modules taking turns to interact with the environment [11]. From another perspective to solve CMAB problem, portfolio greedy search method searches scripts rather than actions in multi-unit combat scenarios [7]. Moreover, Strategy Creation via Voting (SCV) algorithm uses a voting method to generate a large set of novel strategies from existing expert-based ones and use them in games [14]. All these search-based methods mentioned above need inner models of games to do some simulations so as to get evaluations of states required by their algorithms. However, in practice, it is impossible to acquire the model when put in the environment with which we are not familiar. To handle this situation, in this work, we present a model-free MARL approach.

2.2 Multi-agent Reinforcement Learning Methods

Although independent deep Q-networks (DQN) makes big achievements on Atari games [5], its performance on multi-agent systems is not quite competitive. Altering the Actor-critic architecture [15], counterfactual multi-agent policy gradient (COMA) presents a centralised critic for estimation of a counterfactual advantage for decentralised policies [16]. Like COMA, multi-agent deep deterministic policy gradient (MADDPG) adds observations and actions of all agents in centralised critic to improve cooperation and competition [17], which is infeasible when impossible to have the model. Moreover, Multi-agent bidirectionally-coordinated network (BiCNet) presents a vectorized actor-critic framework to learn better coordination strategies [18]. However, none of These approaches can handle situations where the number of the units will vary in one game. The main drawback may lie in the poor scalability of the action-value function which is also called Q-function. Most recently, mean field multi-agent reinforcement learning (MFRL) is proposed to tackle the multi-agent reinforcement learning problems when a large and variable number of agents co-exist [12]. Unfortunately, MFRL needs all agents to be homogenous, which is a strong limitation to apply to other scenarios. In this paper, we extend the work above and try to explore a more general architecture for RTS games.

3 Preliminary

In this section, we will mainly introduce the background of the basic theories about MARL and related academic fields.

3.1 Stochastic Game and Reinforcement Learning

A stochastic game is a model for dynamic games repeatedly played by one or more players with uncertainty of the next game state [19]. This model can be generally adapted to the MARL by adding extra ingredients. An N-agent stochastic game is a Markov Decision Process (MDPs) defined by a tuple

$$\Gamma \triangleq \langle \mathcal{S}, \mathcal{A}^1, \ldots, \mathcal{A}^N, \mathcal{U}, p, r^1, \ldots, r^N, N, \gamma \rangle$$

where $\mathcal{S}$ denotes the state space; $\mathcal{A}^i$ is the action space of agent $i \in \{1, \ldots, N\}$; $U = \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$ is the joint actions space; p denotes the transition function: $\mathcal{S} \times \mathcal{U} \times \mathcal{S} \rightarrow [0, 1]$; The individual payoff function r_i gives utilities to each players participated in the game, which is the immediate reward in RL: $\mathcal{S} \times \mathcal{A}^i \rightarrow \mathbb{R}$, $i \in \{1, \ldots, N\}$; The constant $\gamma \in [0, 1)$ denotes the reward discount, which can serve as the terminator in the infinite-time-step scenarios (Fig. 1).

Unlike independent RL, there are more than one agent interacting with the environment simultaneously in MARL. Given the state, each agent takes one action every timestep according to their policy π^{i}: $\mathcal{S} \rightarrow \Omega(\mathcal{A}^i)$, in which Ω is the probability over agent i's action space $\mathcal{A}^i$. After taking in the joint action of all the agent, the

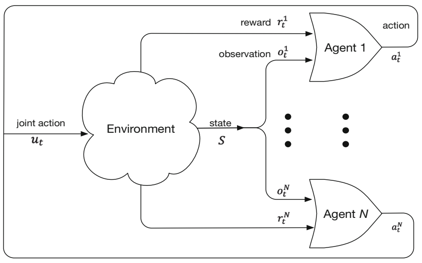

Fig. 1. Interaction between the agents and the environment in multi-agent scenario

environment produces the next state and gives the feedbacks for each agent. Each agent's cumulative rewards G_t^i in one episode can be written as:

$$G_t^i \triangleq \sum\nolimits_{k=0}^{\infty} \gamma^k r_{t+k+1}^i \tag{1}$$

Thus, the state-value function and the action-value for policy π can be respectively defined as:

$$\begin{gathered} v_\pi^i(s) \triangleq \mathbb{E}_\pi[G_t|s], \text{and} \\ q_\pi^i\left(s, a_t^i\right) \triangleq \mathbb{E}_\pi\left[G_t|s, a_t^i\right] = \mathbb{E}_p\left[r_t^i + v_\pi^i(s')\right] \end{gathered} \tag{2}$$

where s' is the next state transferred from s according to the probability p at time t.

3.2 Policy Gradient Theorem and Actor-Critic Architecture

Policy Gradient (PG) method is one branch of reinforcement learning. By this method, the policy can be parameterized as $\pi_{\boldsymbol{\theta}}(a|s)$ if π is differentiable with respect to its parameters $\boldsymbol{\theta}$ [1]. More specifically, the goal of PG method is to figure out the optimal stochastic policy $\pi_{\boldsymbol{\theta}}^*$: $\mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ which maximizes the performance measure, $J(\boldsymbol{\theta}) \triangleq v_{\pi_{\boldsymbol{\theta}}}(s_0)$, and its gradient is proved to be:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi,p}[G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)] \tag{3}$$

Equation (3) is the core formula of REINFROCE algorithm [20]. Similarly, using a parameterized baseline $\widehat{v}_{\boldsymbol{w}}(s)$ to significantly reduce the variance of the gradient, PG methods have been applied to modern actor-critic architecture:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi,p}[(G_t - \widehat{v}_{\boldsymbol{w}}(s)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)] \tag{4}$$

where $\widehat{v}_{\boldsymbol{w}}(s)$ can be viewed as the critic and $\pi_{\boldsymbol{\theta}}(a|s)$ is called the actor. Sometimes, the critic also can be replaced by the action-value function $q_{\boldsymbol{w}}(s,a)$, and the term $G_t - \widehat{v}_{\boldsymbol{w}}(s)$ can be changed to:

$$Adv = q_{\boldsymbol{w}}(s, a) - \widehat{v}(s) \tag{5}$$

4 Methods

We mainly discuss our methods in three aspects in this section. Firstly, we will introduce our scalable architecture. Secondly, we design the reward function to fit into this architecture. Finally, we present our algorithm in the architecture, particularly for RTS games.

4.1 The Architecture

The proposed MARL architecture follows the paradigm of centralized training and decentralized executing [21], which can be integrated into the actor-critic methods, where units (both homogeneous and heterogenetic) play the role of the actor and an evaluation function play the role of the critic. Two procedures iteratively go through during the whole learning process, which are executing and training. The critic is only used in the training. During the executing phase, each actor follows its own policy π_i to simultaneously interact with the environment and save samples respectively in the memory buffer. While in the training phase, the critic reads samples from the memory and train the network (Fig. 2).

Many deep learning architectures related to MARL failed to handle the situation where agents are heterogenetic, or the number of the units varies from time to time in one game episode. This may root in the intractable problem of the poor scalability of the widely-used parameterized action-value function. Because $Q(s,a)$ needs the action of agents, but units will be created or killed, which causes the input module of the Q network collapse. Another reason why those approach hardly work in RTS games may be that, different kinds of units have different action spaces, which requires multiple Q functions, resulting in redundant trainings.

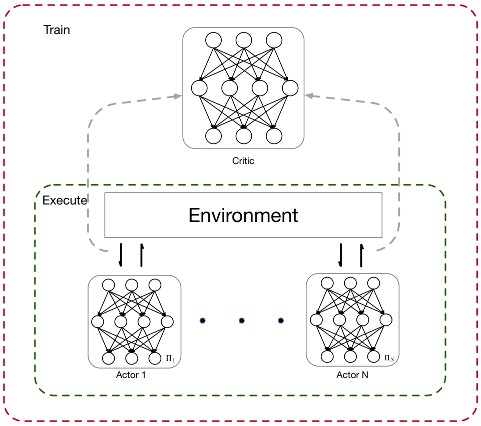

Fig. 2. Learning architecture

Therefore, to avoid adding actions in deep neural network, in our architecture, we just use the state-value function $V(s)$ as the critic, and alter the Eq. (5) by

$$\delta = r_t + \gamma v(s') - v(s) \tag{6}$$

in which δ is called Temporal-Difference (TD) error. It is worth to note that, unlike other approaches which uses multiple critics for actors, we simply use one and the only global state-value critic to evaluate how good states are. This global state-value function is shared by all actors during training, which can be generally adapted to both cooperation and competition scenarios. It could work because units from one player usually make actions to stay at states which have the higher value. Apart from this, we regard instances of different types of agents as the actors. Individual units on the map in RTS games could be viewed as instances or actors. In our setting, actors of the same type share experiences and parameters to accelerate the training process.

To make the architecture complete, as Fig. 3 shows, we also design the deep neural network framework. The encoded spatial features first go through the shared layers for both critic and actor network, then the output flows to the heads of critic and actor respectively. The parameters of all types of units are initiated as different modules ahead in actor network. In order to avoid all units following the same trajectory, we concatenate the encoding of the vary unit's information (including their location and attributes) on the game map into the shared outputs upstream, and the actor network activate the module according to the unit type.

4.2 Reward Design

Not only RL agents learn from rewards, but also the ultimate target of RL is to maximize the expected cumulative rewards. It is of great importance to set up an appropriate reward function for learning in multi-agent systems. Inspired by BiCNet [18], we establish our global reward function as follows.

Using the notations in Sect. 3.1, we define the hit points (hp) of all units of player p at timestep t as $HP_t^p \triangleq \sum_i hp_t^i$, where hp_t^i stands for the hp of agent i; Likewise, on the other hand, $HP_t^{-p} \triangleq \sum_{-i} hp_t^{-i}$ represents for the opponent's situation. The global reward function for player p can be designed by:

$$R_{\Delta t}^p = \Delta HP_{\Delta t}^p - \Delta HP_{\Delta t}^{-p} \tag{7}$$

where

$$\Delta HP_{\Delta t}^p = \left(HP_t^p - HP_{t'}^p\right) - \left(HP_t^{-p} - HP_{t'}^{-p}\right) \tag{8}$$

Equation (8) indicates that sampling starts at time t' and ends at time t because the action last for Δt. From Eq. (7), we can easily get $R_{\Delta t}^p + R_{\Delta t}^{-p} = 0$, which means this reward function leads to a two-player zero-sum game.

We take this global reward function rather than other shaped reward or individual reward for several considerations: (1) The function is simple and general for RTS multi-agent systems. In addition, it is very flexible for that we can easily change rewards for all units by simply changing Eq. (8); (2) It meets the requirement of maximizing the team's utility. In other words, the goal of player *p*'s units is to increase team's common interests rather than themselves', which allows units to sacrifice for team's benefit. (3) In our architecture, this function can be adapted to both cooperation and competition scenarios. The reward generated by environment is shared by all units from one side. Correspondingly, we also use one critic shared by all actors, which naturally combines the reward function and the architecture together.

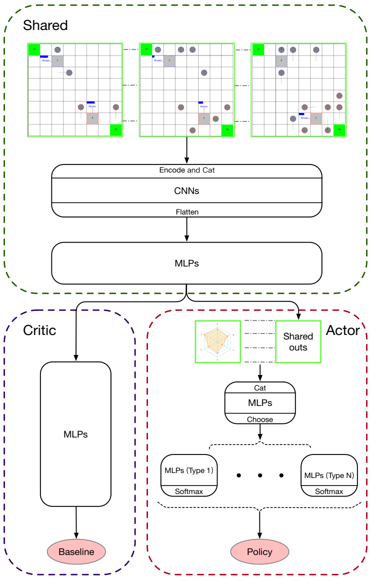

Fig. 3. Network architecture of our approach

4.3 Policy Gradient for Semi-MDP

Markov Decision Process (MDP) is a classic mathematical model for reinforcement learning. In this model, actions are taken in consecutive timesteps with immediate rewards coming afterwards. In fact, situation is much more complicated where rewards are received when actions have been last for a while. This closed-loop policies for taking action over a period of time is called *options* [23], which extends MDP to Semi-MDP (SMDP) (Fig. 4).

The feature mentioned above regarding SMDP implies that not in every single timestep will the unit receive a meaningful reward from environment and different actions takes different time to finish, which indicates that units may asynchronously take their actions and receive their rewards. To tackle this problem, we propose Semi-MARL Policy Gradient (SPG) for multi-agent learning in SMDP.

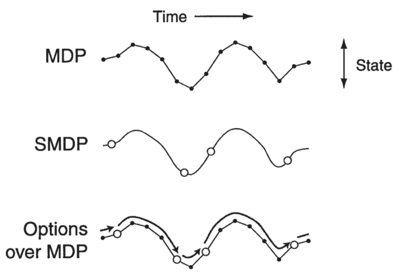

Fig. 4. Difference between MDP and SMDP [23]

Using the notation previously defined, the policies or the options of M heterogenetic agents in game can be parameterized by $\Theta = \{\theta^1, \cdots, \theta^M\}$, $\Pi = \{\pi^1, \cdots, \pi^M | \pi^i = \pi_{\theta^i}\}$, and the gradient for agent i in SMDP model can be written as:

$$\nabla_{\theta^i} J(\theta^i) = \mathbb{E}_{a \sim \pi_{\theta^i},\, s \sim p}\left[\delta_{t \to t'} \nabla_{\theta^i} \log \pi_{\theta^i}\left(a_t^i | s_t\right)\right] \tag{9}$$

Here we call $\delta_{t \to t'} = r^i\left(s_t, a_t^i\right) + \gamma^{t'} v(s_{t'}) - v(s_t)$ as *Semi-Temporal-Difference* (STD) error, where $t' > t$ denotes that sampling begins at time t and terminates at t'.

Furthermore, samples are recorded as tuples of $\langle k, s, a, r, s' \rangle$, where k denotes the type of units, s and s' denotes the beginning state and ending state respectively, a represents the action taken at the state, and r is the delayed reward received when doing action at state s and finished at state s'. During training, as Fig. 3 shows, samples are sent into different actors according to k and the loss of the centralized state-value critic parametrized by ω is defined as:

$$\mathcal{L}(\omega) = (y - v_\omega(s))^2 \tag{10}$$

where

$$y = \begin{cases} r + \gamma^k v_\omega(s') & \text{for terminal} \quad \mathrm{s}' \\ r & \text{for non terminal} \quad \mathrm{s}' \end{cases} \tag{11}$$

If ω is differentiable, we can get the following term for optimizers to update:

$$\nabla_\omega \mathcal{L}(\omega) = (y - v_\omega(s)) \nabla_\omega v_\omega(s') \tag{12}$$

As for the update for actor i, the optimizing target is to maximize the performance $J(\theta^i)$ or minimize the negative performance, and differentiating it gives

$$\nabla_{\theta^i} \mathcal{J}(\theta^i) = -\nabla_\theta^i J(\theta^i) = -\delta \nabla_{\theta^i} \log \pi_{\theta^i}(a|s) \tag{13}$$

Algorithm 1 Semi-MARL Policy Gradient for RTS Games

```
Initialize critic parameters $\omega$
Initialize actor parameters $\theta^1, \cdots, \theta^M$
Initialize counter T = 0
**Input:** Initial state s

**procedure** ASSIGN ACTION(State s, Player p)
    Sample actions for free units player p has at state s based on $\Pi$
    **return** sampled actions

**procedure** SELF PLAY
    **while** $T < T_{max}$ **do**
        $\mathcal{A}_1 \leftarrow$ **ASSIGN ACTION**$(s, 1)$
        $\mathcal{A}_2 \leftarrow$ **ASSIGN ACTION**$(s, 2)$
        Take actions $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and wait for actions to finish
        Receive next state $s^{'}$
        Receive rewards **r** asynchronously and records samples    ▷ In backend
        Divide samples into different parts according to unit type
        Use Eq.(12) and Eq.(13) to update $\omega$ and $\Theta$
        $s \leftarrow s^{'}$
        $T \leftarrow T + 1$
```

5 Experiments

5.1 Environment

We choose MicroRTS as the main body of the study. MicroRTS is a mini real-time strategy game implemented in Java. To make the environment work for reinforcement learning, we add the interfaces for Python so as to wrap the environment like gym [13]. The motivation of MicroRTS is to strip the RTS logic from large games like StarCraft to improve the performance of the system and reduce the work of engineering so that experiments can be quickly conducted on it.

The environment of the game is shown in Fig. 5, where the entire battlefield is divided into $m \times n$ grids, each of which can only be occupied by one unit every timestep. There are seven different types of units, which are respectively Base, Worker, Barracks, Heavy, Light, Ranged, Resource. Different types of units have different attributes and behaviors: Base can only produce Worker; Worker can collect resources for production and construction tasks; Barracks are responsible for the production of combat types of units; Heavy and light are melee

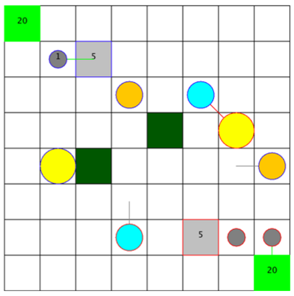

Fig. 5. Screenshot of MicroRTS

units (they can only attack the enemy directly adjacent to them); Ranged is a ranged attack unit; Resource is a neutral unit that is consumed by construction and production tasks. The game is two-player and different side of the units is of different colors. In this game, each player's goal is to destroy all enemy units in a limited time.

Like other RTS games, MicroRTS also has the following challenges:

- Long-term and short-term planning: similar to the real world, many cause-and-effect relationships are not clear at a glance; Decisions made earlier in the game may take a few steps to show effects.
- Real-time: unlike chess in which two players take turns to play, MicroRTS requires the player to make decisions on all units of his in 100 ms.
- Large action space: as the number of units increases, the number of action combinations will increase exponentially.

5.2 Experiment Setup

In our experiment, we mainly evaluate micro-management of our AI which is trained by the approach we proposed. Experiments are conducted in the following settings:

- 1 VS 1: The map size is 4 $\times$ 4, each player only has one Light to fight for them. The max cycle of game is 1000 timesteps. No war fog on the battlefield (perfect information).
- 2 VS 2: The map size is 4 $\times$ 4, each player has one Light and one Heavy. The max cycle of game is 1000 timesteps. No war fog on the battlefield.
- 6 VS 6: The map size is 6 $\times$ 6, each player can control six Lights and totally 12 units. The max cycle of game is 1500 timesteps. No war fog on the battlefield.

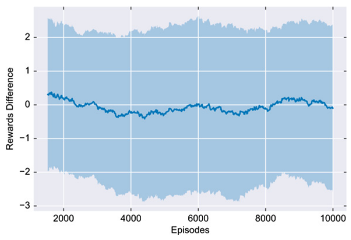

Fig. 6. Rewards difference

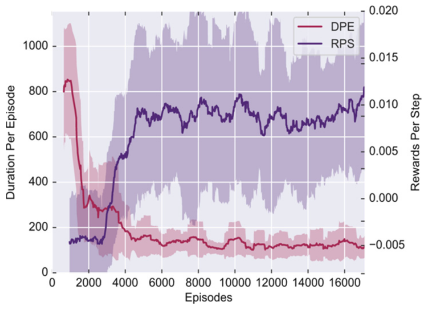

Fig. 7. Rewards and duration curve (6 VS 6)

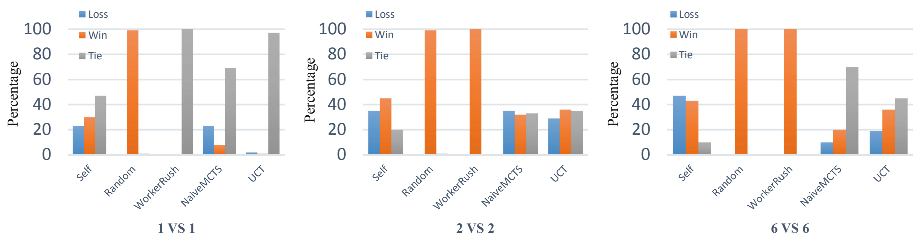

Fig. 8. Win, loss, tie rates of 1 VS 1(left), 2 VS 2 (middle) and 6 VS 6 (right) against the baseline AIs (Random, WorkerRush, NaiveMCTS and UCT) respectively

All experiments above are carried out in self-play manner with multiple threads and we train a single network both for player 1 and player 2. We use no abstract actions with human knowledge, but we use translated actions. Translated action functions like that if unit take "left probe" and there is an enemy, then the action will be translated to "attack left"; if there is an empty box, then the action will be translated to "move left". After training, four baseline AIs are selected to combat with our AI to measure the performance. The four baseline AIs we choose are Random, UCT with UCB1 [24], NaiveMCTS, and WorkerRush [25]. NaiveMCTS and UCT are search-based and WorkerRush is script-based. Random is included to test the generalization ability of our neural network since Random robot can stochastically bring various states, which is necessary because some AIs can defeat skillful opponents but fail to win the weaker one they have never seen before. Script-based robot WorkerRush is chosen as it is known to perform well in MicroRTS. UCT is selected because it is the classic and the traditional search-based MCTS algorithm. Additionally, we choose NaiveMCTS because it performs significantly better than other search-based algorithms as the number of units increases [11]. These search-based AI mentioned above are quite strong in small scenarios.

5.3 Experiment Results

During training, we log the rewards difference between player 1 and player 2 every episode to see if anyone takes advantage of the other or gives up. As Fig. 6 shows, the curve representing rewards difference is around zero over time. It indicates that both players parameterized in one network in our self-play settings struggle to get more rewards. Figure 7 shows the 6 VS 6 curve of the duration of battle and the average rewards received for both two players over 16000 episodes. We can infer that, both two players we trained tend to win games as fast as possible and get as much rewards as they can.

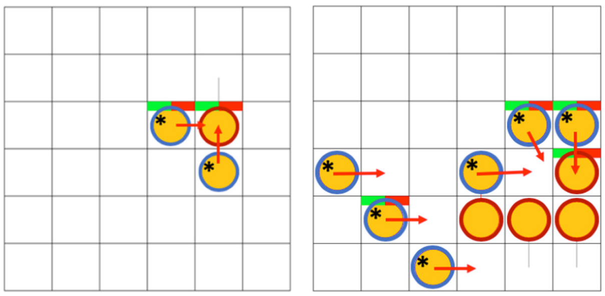

Fig. 9. Learning to cooperate after training in 6 VS 6 settings. Units with blue profile are ours (also marked with '*') and the ones with red profile are enemies'. The red arrows extend from units indicate trends of the next move. (Color figure online)

After training, we select our best model to combat with the baseline AIs and record the average rate of loss, win and tie in 100 games respectively. If one of the two players eliminates all units of the other, then he wins and if no one defeats his opponent in a limited time, then they tie. Figure 8 shows the result of our AI's performance. As we can see, in the 1 VS 1 setting, the result of battles against other AIs is mostly tie except Random. However, in the setting of 2 VS 2 which needs units to cooperate with each other, our AI can completely defeat WorkerRush which is a strong script-based AI. Furthermore, in the 6 VS 6 experiment, the AI's winning rate surpasses the search-based AIs (NaiveMCTS and UCT) for the first time. The reason why our AI does better in larger scenarios (2 vs 2 and 6 vs 6) may be that the search-based algorithm only has 100 ms (because of real-time strategy) for them to search the game. In all, our AI is highly competitive to these baseline AIs.

To visualize whether our AI learns to cooperate with each other, we capture the frames during game. As Fig. 9 shows, the AI is able to chase the enemy and make a joint attack on it (left). Furthermore, it also learns to siege the enemies in the corner to eliminate them (right).

6 Conclusion and Future Work

In this paper we firstly proposed a general and scalable reinforcement learning architecture for multi-agent systems and make it work for semi-MDP model. We then design the reward function for this architecture and present our algorithms. We mainly use MicroRTS environment to evaluate the micro-management performance of our algorithm. The result shows that the AI trained by our approach is highly competitive. For the future work, we are going to test the full game performance and we observe that the neural network might be the bottleneck of our algorithm and further experiments about how networks should be better built and the better tuning of the network hyperparameter is worth to be carried out.

Acknowledgement. This work is supported by the national key research and development program of China under grant No. 2019YFB2102200, National Natural Science Foundation of China under Grant No. 61672154, No. 61972086, and the Fund of ZTE Corporation under Grant No. HC-CN-20190826009.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
2. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
3. Vinyals, O., et al.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)
4. Berner, C., et al.: Dota 2 with large scale deep reinforcement learning (2019)
5. Mnih, V., et al.: Playing atari with deep reinforcement learning (2013)
6. Ontanón, S., Buro, M.: Adversarial hierarchical-task network planning for complex real-time games. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
7. Churchill, D., Buro, M.: Portfolio greedy search and simulation for large-scale combat in StarCraft. In: 2013 IEEE Conference on Computational Intelligence in Games (CIG). IEEE (2013)
8. Churchill, D., Saffidine, A., Buro, M.: Fast heuristic search for RTS game combat scenarios. In: Eighth Artificial Intelligence and Interactive Digital Entertainment Conference (2012)
9. Marino, J.R.H., et al.: Evolving action abstractions for real-time planning in extensive-form games. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33 (2019)
10. Barriga, N.A., Stanescu, M., Buro, M.: Combining strategic learning with tactical search in real-time strategy games. In: Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference (2017)
11. Ontanón, S.: The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference (2013)
12. Yang, Y., et al.: Mean field multi-agent reinforcement learning (2018)
13. Brockman, G., et al.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
14. Silva, C.R., et al.: Strategy generation for multi-unit real-time games via voting. IEEE Trans. Games **11**, 426–435 (2018)
15. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning (2016)
16. Foerster, J.N., et al.: Counterfactual multi-agent policy gradients. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
17. Lowe, R., et al.: Multi-agent actor-critic for mixed cooperative-competitive environments. In: Advances in Neural Information Processing Systems (2017)
18. Peng, P., et al.: Multiagent bidirectionally-coordinated nets: emergence of human-level coordination in learning to play starcraft combat games (2017)
19. Shapley, L.S.: Stochastic games. Proc. Nat. Acad. Sci. **39**(10), 1095–1100 (1953)
20. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn. **8**(3-4), 229–256 (1992)
21. Oliehoek, F.A., Spaan, M.T.J., Vlassis, N.: Optimal and approximate Q-value functions for decentralized POMDPs. J. Artif. Intell. Res. **32**, 289–353 (2008)

22. Kraemer, L., Banerjee, B.: Multi-agent reinforcement learning as a rehearsal for decentralized planning. Neurocomputing **190**, 82–94 (2016)
23. Sutton, R.S., Precup, D., Singh, S.: Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**(1-2), 181–211 (1999)
24. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_29
25. Ontañón, S., et al.: The first microrts artificial intelligence competition. AI Mag. **39**(1), 75–83 (2018)