Master Multiple Real-Time Strategy Games with a Unified Learning Model Using Multi-agent Reinforcement Learning

Bo Ling[1], Xiang Liu[1], Jin Jiang[1(✉)], Weiwei Wu[1], Wanyuan Wang[1], Yan Lyu[1], and Xueyong Xu[2(✉)]

[1] Southeast University, Nanjing, China
boling_william@163.com, {xiangliu,weiweiwu,wywang,lvyanly}@seu.edu.cn, analysis.jinger@gmail.com
[2] North Information Control Research Academy Group Co., Ltd., Nanjing, China
xxyyeah@163.com

Abstract. General artificial intelligence requires an intelligent agent to understand or learn any intellectual tasks like a human being. Diverse and complex real-time strategy (RTS) game for artificial intelligence research is a promising stepping stone to achieve the goal. In the last decade, the strongest agents have either simplified the key elements of the game, or used expert rules with human knowledge, or focused on a specific environment. In this paper, we propose a unified learning model that can master various environments in RTS game without human knowledge. We use a multi-agent reinforcement learning algorithm that uses data from agents in a diverse league played on multiple maps to train the deep neural network model. We evaluate our model in microRTS, a simple real-time strategy game. The results show that the agent is competitive against the strong benchmarks in different environments.

Keywords: Real-time strategy game · Multi-agent reinforcement learning · League learning · Unified model

1 Introduction

Game AI has a history of active research for decades with games often serving as challenge problems, benchmarks and milestones for progress. Muzero, a model-based reinforcement learning agent, masters Go, chess shogi and Atari without rules. Pluribus, a superhuman AI masters six-player poker [14]. OpenAI Five defeated a team of professional Dota 2 players and 99.4% of online players [5]. AlphaStar was rated at Grandmaster level for all three StarCraft races and above 99.8% of officially ranked human players [4]. Tencent developed a superhuman

B. Ling and X. Liu—These authors contribute equally.

H. Zhang et al. (Eds.): NCAA 2022, CCIS 1638, pp. 27–39, 2022.
https://doi.org/10.1007/978-981-19-6135-9_3

AI agents that can defeat top esports players on a popular MOBA game, Honor of Kings [18].

To the goal of general artificial intelligence, an intelligence agent is expected to understand or learn any intellectual tasks like a human being. Although these AIs are greatly successful, they have a major limitation, i.e., only suitable for a specific environment. Trying to explore an agent that can master various environments is meaningful and valuable.

In this paper, we study one of the most challenging domains, real-time strategy games. It has real-time, simultaneous, huge action space, huge state space, and multi-agent characteristics. Specifically, we study an algorithm to master multiple real-time strategy games. Multiple games means that the game environment is different in the size of the map, the initial position of the unit, the number of resources, and so on. In different environments, completely different strategies are usually required. Script-based or search-based algorithms usually can be applied to multiple maps but rely on expert experience and domain knowledge. In this paper, we explore a unified learning model applicable to multiple environments in real-time strategy game. We summarize the contributions of this paper as follows.

- To the best of our knowledge, we are the first to explore a unified learning model to play multiple real-time strategy games.
- Under the actor-learner pattern, we design a general neural network model, including the encoding of multi-model inputs and the decoupling of action outputs. The model is suitable for the input of various maps and units, and the output of various action types.
- To improve robustness, we explore a multi-agent reinforcement learning algorithm to enable effective explorations for multi-agent competitive environments.
- We choose three different maps for experiments in the domain of microRTS, a simple RTS game developed for research purposes. Experimental results demonstrate that various environments can apply our proposed model. In addition, the trained AI agent outperforms benchmarks in all maps.

The rest of the paper is organized as follows. Section 2 reviews the related work. In Sect. 3, we introduce the model and formulate the Reinforcement Learning problems. In Sect. 4, we propose a unified learning model to play multiple real-time strategy games. We then conduct experiments in Sect. 5 to validate the performance of the proposed method. Finally, we conclude the paper and discuss the future work in Sect. 6.

2 Related Work

The research methods of RTS games mainly include search-based, rule-based and learning-based methods. In this section, we mainly review these three methods separately.

2.1 Rule-Based Methods

Rule-based research compiles the rules summarized by human players in practice into programs, and the game program selects the corresponding strategy execution according to the game situation during the game.

Silva et al. [16] proposed Strategy Creation via Voting (SCV) [15] by using a voting method to generate a set of strategies from existing expert strategy and an opponent modeling scheme to choose appropriate strategies from the generated pool of possibilities. Marino et al. [11] and others constructed a new script pool by combining and replacing expert strategy scripts, using genetic algorithms to select a collection of scripts with a high winning rate.

The main drawback of rule-based methods is that they only work on a limited amount of expert-designed strategies. This led to rigid scripted tactical micromanagement. By contrast, we design league training so that the trained AI can effectively explore various strategies.

2.2 Search-Based Methods

Churchill and Buro [7] proposed an Alpha-Beta Considering Durations (ABCD) [11] algorithm based on min-max search and Alpha-Beta pruning. Considering durative actions, the alpha-beta algorithm implements a tree alteration technique to address simultaneous actions. Ontañón [13] introduced NaïveMCTS, which uses a sampling strategy to exploit the tree structure of Combinatorial Multi-Armed Bandits. Yang and Ontanón [20] investigate to combine scripts into the tree policy for the convergence guarantees of MCTS. However, these tree-search based methods usually depend on an efficient forward model, which requires internal models of games to examine the outcome of different combinations. Instead, we use a model-free reinforcement learning approach.

In addition, due to real-time strategy games have a huge search space, the search algorithms described above require a lot of time. The following algorithms try to limit the search space. Puppet Search (PS) [7] defines a search space for the values of script parameters. Strategy Tactics [3] combines PS's search in the script-parameter space with a NaïveMCTS search in the original state space for the combat units. Moraes and Lelis [12] introduced a search algorithm based on combinatorial multi-armed bandits (CMABs) that used asymmetric action abstractions schemes to reduce the action space considered during search [10]. Lelis [12] proposed Stratified Strategy Selection (SSS) to micromanage units in RTS games, which divides players' units into types under the assumption that units with the same type follow the same script. Lin et al. [9] further took into account the time constraint when generating the sub-game tree and introduced the adversarial player as the opponent strategy.

In these methods, scripts are also used to guide the search. The action-abstracted space usually restricts the behaviour of agents, as such space tends to be limited to a small number of handcrafted strategies.

2.3 Learning-Based Methods

Recently, multi-agent reinforcement learning methods for specific scenarios has been studied in depth. Low et al. [10] proposed multi-agent deep deterministic policy gradient (MADDPG) [2], an adaptation of actor-critic methods that can be used in competition and cooperation scenarios. Since the proposed model requires each agent to have its own independent critic network and actor network, this model is limited to micromanagement tasks with a fixed number of input units.

In 2019, OpenAI introduced OpenAI Five [5] for playing 5v5 games in Dota 2, which can defeat a team of professional Dota 2 players. The OpenAI Five applies deep reinforcement learning via self-play. Recently, Tencent developed AI programs towards playing full 5v5 Multiplayer Online Battle Arena games (MOBA) in Honor of Kings through curriculum self-play learning, which can defeat top esports players [18]. As a sub-problem of the RTS games, there is no need to consider building base, building barracks, training different types of units and so on for MOBA games. Therefore, these models do not consider the situation that the number of agents may change with the progress of the game, i.e., new units may be produced and existing units may be eliminated. Our approach can handle the problem because we toward a full real-time strategy game instead of just micromanagement.

In 2019, DeepMind developed an AI in the full game of StarCraft II, called Alphastar, with the ability to reach the Grandmaster level for all three StarCraft races. The AI is trained through end-to-end multi-agent reinforcement learning. The main difference between our model and AlphaStar is that we explore playing games on various different scale of maps instead of playing games in specific scenarios.

3 Preliminary

3.1 Real-Time Strategy Games

The RTS game is essentially a simplified military simulation. Combat scenarios in RTS games can be described as finite zero-sum two-player games with simultaneous and durative moves defined by a tuple $(\mathcal{N}, \mathcal{S}, s_{\text{init}}, \mathcal{U}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \pi)$,

- $\mathcal{N} = \{i, -i\}$: the set of players.
- $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$: the set of states, where $\mathcal{D}$ is the set of non-terminal states and F is the set of terminal states.
- $s_{init} \in \mathcal{D}$: the start state of the games.
- $\mathcal{A} = \mathcal{A}_{\text{i}} \times \mathcal{A}_{-i}$: the set of joint actions.
- $\mathcal{A}_{\text{i}}(s)$: the set of legal actions that player i can perform in state s.
- $\mathcal{U} = \mathcal{U}_{\text{i}} \times \mathcal{U}_{-i}$: the set of joint units.
- $\mathcal{U}(s)$: the set of available units in state s.
- $\mathcal{R}_i : \mathcal{F} \to \mathbb{R}$ is a utility function with $\mathcal{R}_i(s) = -\mathcal{R}_{-i}(s)$ for any $s \in \mathcal{F}$.
- $\mathcal{P} : \delta \times \mathcal{A}_{\text{i}} \times \mathcal{A}_{-i} \to \mathcal{S}$ determines the successor state for a state s and a set of join actions taken at s.
- $\pi(\mathcal{A} \mid \mathcal{S}) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the policy of a player.

3.2 Reinforcement Learning

We consider the Reinforcement Learning problem in a Markov Decision Process (MDP) denoted as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma, T)$ [6]. Here, $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function, γ is the discount factor, and T is the maximum episode length. The goal is to maximize the expected discounted return of the policy:

$$\mathbb{E}_{\tau}\left[\sum_{t=0}^{T-1} \gamma^{t} r_{t}\right] \tag{1}$$

where τ is the trajectory $(\mathrm{s}_0, \mathrm{a}_0, \mathrm{r}_0, \ldots, \mathrm{s}_{\mathrm{T}-1}, \mathrm{a}_{\mathrm{T}-1}\mathrm{r}_{\mathrm{T}-1})$, $\mathrm{s_t} \sim \mathcal{P}\left(* \mid \mathrm{s}_{\mathrm{t}-1}, \mathrm{a}_{\mathrm{t}-1}\right)$ and $\mathrm{a_t} \sim \pi_{\theta}\left(* \mid s_t\right), r_t = r\left(s_t, a_t\right)$.

4 Methods

In this section, we introduce our methods from three aspects. Firstly, we introduce our neural network architecture for microRTS. Secondly, we describe the design of league learning used to generate training data. Finally, we present our multi-agent reinforcement learning algorithm.

4.1 The Architecture

We use an actor-critic network structure, where actor refers to policy network, and critic refers to value network. The network follows the paradigm of centralized training and decentralized executing, i.e., the critic network is only used for training, and the actor-network is used for training and execution. In our design, all units share a policy network and a value network. Specifically, the policy network calculates the policy for each unit independently, using the global state information and the information of an input unit, and the value network evaluates the global state value of a player. Based on the deep neural network method, we need to design a DNN-represented policy $\pi_{\theta}\left(a_t \mid s_t\right)$ with parameters θ. Its inputs include previous observations, current units of a player and the information of the unit types. Its output are actions and the game state value. In order to provide informative observations to the actor, we develop multi-modal features, including a comprehensive list of both scalar and spatial features. The scalar feature is consisted of units' type attributes, e.g., health point (hp), relative position, speed, action types, etc. The spatial features include convolutional channels derived from the player's global view, e.g., player resources, the type of units on each grid, and the actions of the units on each grid. To deal with the problem that different types of agents have different action spaces and the explosion of combined action spaces, we develop hierarchical action heads and discretize each head. Specifically, the actor predicts the output actions hierarchically: 1) what action to select, including move, attack, return, harvest, and produce. 2) how to act, e.g., a discretized direction to move. 3) which unit type to produce given an input unit, including worker, barrack, base, light, heavy, ranged. The network structure is shown in Fig. 1.

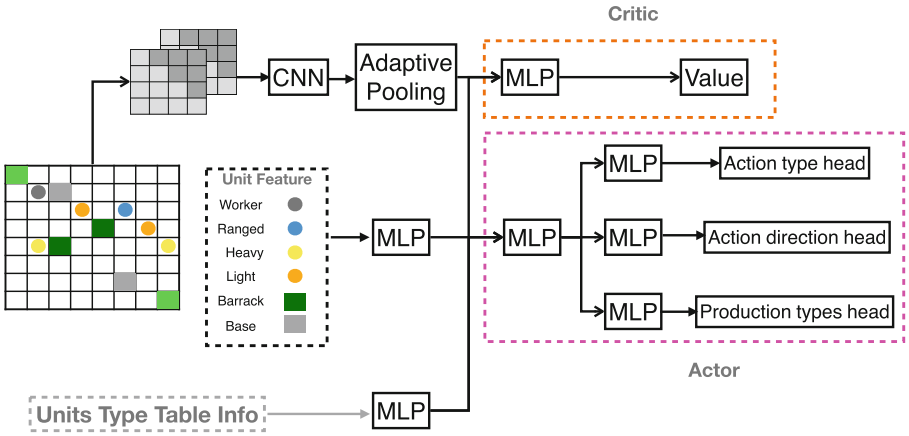


Fig. 1. Network architcture for RTS game

Internally, in order to handle different map sizes, the observations are first convolved and then passed to an adaptive pooling layer to output fixed-length vectors. The adaptive pooling layer removes the fixed size constraint of the network. Therefore, the model could process maps of different sizes. The model concatenates the encoding of the various information of the unit types and the output of the pooling layer into a vector $\boldsymbol{v}_1$. The model concatenates the vector $\boldsymbol{v}_1$ and the encoding unit given by a player into a vector $\boldsymbol{v}_2$. The vector $\boldsymbol{v}_1$ is passed to the multi-layer perceptron of the critic network to calculate the state value. The vector $\boldsymbol{v}_2$ is passed to the multi-layer perception of the actor network to calculate the action.

4.2 League Learning

Balduzzi et al. [1] pointed out that, for the approximately transitive game, self-play generates sequences of agents of increasing strength. However, non-transitive games, such as rock-paper-scissors, may chase cycle and fail to progress. Observing this, AlphaStar computes a best response against a non-uniform mixture of opponents to avoid strategic cycles. In this paper, we use a pool of non-uniform mixed opponents as a league and then train the agent through playing against with the league. By making use of the advantages of the built-in script AI, we selected some AI to join the league according to the strength of AI. In this work, we used the following AIs as opponents:

1. Passive: A hard-coded strategy that passively attacks the nearest unit.
2. Random: A heuristic strategy that randomly selects one of the possible player-actions, but the probability of choosing an attack or harvest action is 5 times higher than the other actions.

3. WorkerRush: A hard-coded strategy that builds a barracks, and then constantly produces "worker" units to attack the nearest target (it uses one worker to mine resources).

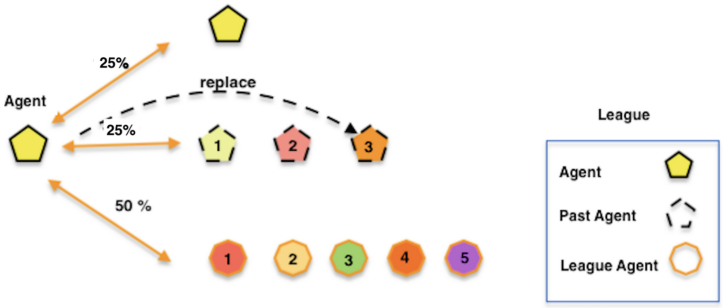


Fig. 2. League training stage three.

In the first stage, the league is set to a weaker version, including Passive and Random. The agent is trained with a proportion of 25% Self-play, 50% against Passive in the league, and an additional 25% of matches against Random in the league. This stage of training ends when the training reaches the 100 thousand timestep or the time to complete the game is stable.

In the second stage, WorkerRush was added to the league. The agent is trained with a proportion of 25% Self-play, 25% against WorkerRush in the league, 25% against Passive in the league, and an additional 25% of matches against Random in the league. This stage of training ends when the training reaches the 100 thousand timestep or the time to complete the game is stable.

In the third stage, we copy the trained agent, named Past Agent, to replace Passive. The agent is trained with a proportion of 25% Self-play, 25% against WorkerRush in the league, 25% against Passive in the league, and an additional 25% of matches against Past Agent. The training process is shown in Fig. 2.

4.3 Multi-agent Reinforcement Learning

We use the actor-critic paradigm [17] to train a value function $v_\theta(s_t)$ with a policy $\pi_\theta(a_t \mid s_t)$. We first introduce the design of reward function. Then, we present our policy gradient update algorithm.

Inspired by Wang et al. [8], we design *Semi-Advantage-Function* (SAF) as follow. Firstly, we calculate the hp of player p at time t as Eq. (2).

$$\mathrm{HP}_t^p \triangleq \sum_i hp_t^i \tag{2}$$

where hp_t^i denotes the hp of agent i at time t. Secondly, we design a single timestep reward of player p as Eq. (3).

$$R_t^p = \left(HP_t^p - HP_{t'}^p\right) - \left(HP_t^{-p} - HP_{t'}^{-p}\right) \tag{3}$$

Thirdly, we define the action reward for unit i of player p at the beginning of the timestep t as Eq. (4).

$$r^p\left(s_t, a_t^i\right) = R_{t+1}^p + \gamma R_{t+2}^p + \cdots + \gamma^{t'-t-1} R_{t'}^p \tag{4}$$

where $t' - t$ is the duration of the unit action. Finally, we design our SAF as Eq. (5).

$$A_t^i = r^p\left(s_t, a_t^i\right) + \gamma^{t'-t} v\left(s_{t'}\right) - v\left(s_t\right) \tag{5}$$

We assume that the action heads are independent to simplify the correlations of action heads, e.g., the production of unit type is conditioned on the action type, which is similar to that of [19]. To perform an action, we select an Action Type with corresponding action parameters. Namely, a_t^i is composed of smaller actions $a_t^{\text{Action Type}|i}$, $a_t^{\text{Action Parameter}|i}$, $a_t^{\text{Produce Type}|i}$. We use Proximal Policy Optimization (PPO) [21] to train the agent. The policy gradient is updated using SAF in the following way (without considering the PPO's clipping for simplicity)

$$\sum_{t=0}^{T-1} \sum_{u \in \mathcal{U}(\mathrm{s_t})} A_t^u \nabla_\theta \log \pi_\theta\left(a_t^{\mathrm{u}} \mid \mathrm{s_t}\right) \tag{6}$$

$$= \sum_{t=0}^{T-1} \sum_{u \in \mathcal{U}(\mathrm{s_t})} A_t^u \nabla_\theta \left(\sum_{d \in D} \log \pi_\theta\left(a_t^{\mathrm{d|u}} \mid \mathrm{s_t}\right) \right) \tag{7}$$

$$= \sum_{t=0}^{T-1} \sum_{u \in \mathcal{U}(\mathrm{s_t})} A_t^u \nabla_\theta \log \left(\prod_{d \in D} \pi_\theta\left(a_t^{\mathrm{d|u}} \mid \mathrm{s_t}\right) \right) \tag{8}$$

where $D = \{\text{Action Type, Action Parameter, Produce Type}\}$. The detail of the proposed method is shown in Algorithm 1.

5 Experiments

5.1 Environment

We choose microRTS as the main body of the research, where microRTS is a small implementation of RTS games, aimed at AI research.

Algorithm 1. The Unified Actor-critic Learning Algorithm for RTS Games

Set initial actor network parameters θ, critic network parameters ω, buffer B, state $s = s_{init}$, update steps T.
Define the policy of league π_{league}.
Define the policy of actor π_{old}.
for iteration: $i = 0, 1, ..., N$ **do**
  for actor: $j = 0, 1, ..., M$ **do**
    Select the policy of the opponent π^j_{league}
    Choose an action for player 1 based on the actor policy $A_1 \leftarrow \pi_\theta old$.
    Choose an action for player 2 based on the critic policy $A_2 \leftarrow \pi^j_{league}$.
    Submit joint actions $A \leftarrow (A_1, A_2)$.
    Update states $s \leftarrow s'$.
    Store the actionable unit u and the unit-selected action a under state s in the form (s, u, a, r, s') in buffer B.
    if $|B| = T$ **then**
      Update parameters θ_{old} according to 6.
    end if
    if $s = s_{end}$ **then**
      $s \leftarrow s_{init}$
    end if
    $T \leftarrow T + 1$
  end for
end for

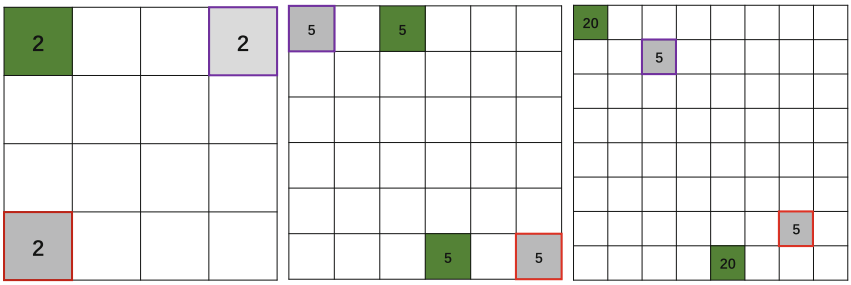
5.2 Experiment Setup

Our RL infrastructure runs on a physical computing cluster. To train our AI model in parallel, we use 8 V100 GPUs and 128 CPUs totally. For each of the experiments, we use 1 GPU and 16 CPUs resource to train. We have selected three maps for experimentation (Fig. 3).

We perform two groups of experiments. One is to train an agent on each map in a self-play manner and evaluate it with benchmarks on the three maps. The other is to train and evaluate an agent on the three maps using the method of league learning. The detailed description of the map is as follows. (1) On the 4×4 map, each player has 1 base and 2 resources. In addition, there are 2 public uncollected resources. The max cycle of game is 1000 timesteps. (2) On the 6×6 map, each player has a base and 5 resources. In addition, there are 5 uncollected resources around each player. The max cycle of game is 1500 timesteps. (3) On the 8×8 map, each player has a base and 5 resources. In addition, there are 20 uncollected resources around each player. The max cycle of game is 2000 timesteps.

5.3 Experiment Results

We use WorkerRush, Passive, Random as the benchmarks to evaluate the performance of our trained agents. To be brief, WorkerRush is an offensive agent eager to win, Passive is a defensive agent who wants to avoid defeat, and Random

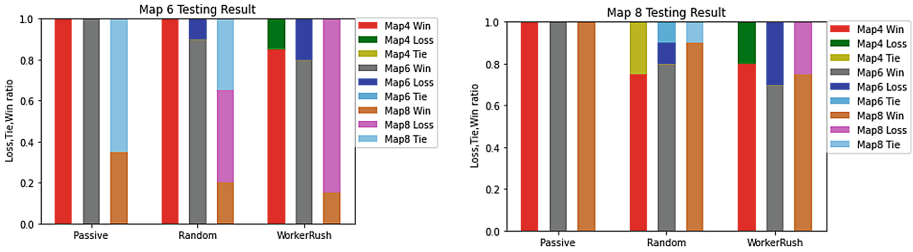


(a) 4×4 Map (b) 6×6 Map (c) 8×8 Map

Fig. 3. Training maps

wants the game to be a tie (more detailed descriptions of these three methods are shown in Sect. 4.2).

In the group of experiments, there are two agents SelfPlay6 and SelfPlay8, which represent agents trained in self-play on 6×6 and 8×8 maps, respectively. The evaluation results are shown in Fig. 4a–4b.



(a) SelfPlay6 AI against with benchmarks (b) SelfPlay8 AI against with benchmarks

Fig. 4. AI against with benchmarks

Figure 4a shows the result of SelfPlay6 agent. We found that SelfPlay6 has a higher winning rate on 6×6 and 4×4 maps, with a winning rate of more than 60%. It is revealed that the agent has a certain generalization performance. However, on the 8×8 map, the winning rate is significantly reduced. By observing the game video, we found that it has obvious offensive intent on the 4×4 map, but not on the 8×8 map. We would evaluate all methods by the metric of Loss, Tie, Win ratio.

Figure 4b shows the result of SelfPlay8 agent. We found that the agent is very competitive against Passive and Random in the three maps, but has a low win rate against WorkerRush. This reveals that the learning strategy of the agent has certain effect, but the strategy is not strong enough. On the one hand, the size of these three maps is not that big, victory mainly depends on micro-management,

and WorkerRush has the advantage of micromanagement. On the other hand, agents trained through self-play may chase cycles resulting in no progress.

Through the results, we find that the AI trained on the medium map has better generalization performance on the small map, while the generalization ability on the large map is weaker. But the trained AI is not strong enough via self-play, thus we examine its performance when combined with league learning.

In the second group of experiments, we train an agent LeagueAllMap, which uses league learning to train on the three maps. The result is shown in Fig. 5.

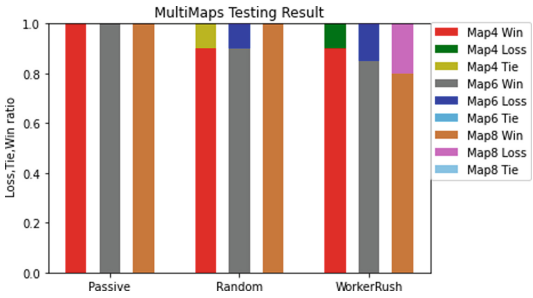


Fig. 5. LeagueAllMap AI against with benchmarks

To improve the generalization and robustness of the agent, we try to make the model play against opponents on three maps at the same time. Furthermore, we use league training to overcome chase cycles in strategy space. Through these methods, the experimental results show that the performance of the agent has been significantly improved. The agent outperforms the benchmarks on all of the three maps.

We found that multi-stage training by setting up opponents with increasing difficulty can effectively improve their abilities. It suggests that this learning method is easier to get rewards from sparse rewards than self-play. Moreover, we find that the model has better results on the three maps due to the simultaneous use of data from three different maps for training.

6 Conclusion and Future Work

In this paper, we develop a unified learning model, which aims to play a full RTS game on multiple maps through deep reinforcement learning. We explored neural networks that can be learned and trained at different scales, and then explored league training methods. To the best of our knowledge, this is the first reinforcement learning based RTS AI program that can play multiple maps and outperform the benchmark.

However, this study still has some limitations. First of all, there is a lack of theoretical proofs for the convergence of simultaneous training on multiple maps. Secondly, the maps we choose to train are not significantly different, and in general, the scale of the map is not that large. Finally, the robustness of our agent needs to be evaluated in comparison with agents with stronger and more diverse strategies. In the future, we will continue explore more diverse and complex maps for experiments.

Acknowledgement. This work is supported by the National Key R&D Program of China under grant 2018AAA0101200, the Natural Science Foundation of China under Grant No. 62102082, 61902062, 61672154, 61972086, the Natural Science Foundation of Jiangsu Province under Grant No. 7709009016 and the Postgraduate Research & Practice Innovation Program of Jiangsu Province of China (KYCX19_0089).

References

1. Balduzzi, D., et al.: Open-ended learning in symmetric zero-sum games. In: International Conference on Machine Learning, pp. 434–443. PMLR (2019)
2. 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)
3. Barriga, N.A., Stanescu, M., Buro, M.: Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In: Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (2015)
4. Berner, C., et al.: Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680 (2019)
5. Brown, N., Sandholm, T.: Superhuman AI for multiplayer poker. Science **365**(6456), 885–890 (2019)
6. Buro, M.: Real-time strategy games: a new AI research challenge. In: IJCAI, vol. 2003, pp. 1534–1535 (2003)
7. Churchill, D., Saffidine, A., Buro, M.: Fast heuristic search for RTS game combat scenarios. In: Eighth Artificial Intelligence and Interactive Digital Entertainment Conference (2012)
8. Konda, V., Tsitsiklis, J.: Actor-critic algorithms. In: Advances in Neural Information Processing Systems 12 (1999)
9. Lin, S., Anshi, Z., Bo, L., Xiaoshi, F.: HTN guided adversarial planning for RTS games. In: 2020 IEEE International Conference on Mechatronics and Automation (ICMA), pp. 1326–1331. IEEE (2020)
10. Lowe, R., Wu, Y.I., Tamar, A., Harb, J., Pieter Abbeel, O., Mordatch, I.: Multi-agent actor-critic for mixed cooperative-competitive environments. In: Advances in Neural Information Processing Systems 30 (2017)
11. Marino, J.R., Moraes, R.O., Toledo, C., Lelis, L.H.: Evolving action abstractions for real-time planning in extensive-form games. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 2330–2337 (2019)
12. Moraes, R., Lelis, L.: Asymmetric action abstractions for multi-unit control in adversarial real-time games. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)

13. 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)
14. Schrittwieser, J., et al.: Mastering atari, go, chess and shogi by planning with a learned model. Nature **588**(7839), 604–609 (2020)
15. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
16. Silva, C., Moraes, R.O., Lelis, L.H., Gal, K.: Strategy generation for multiunit real-time games via voting. IEEE Trans. Games **11**(4), 426–435 (2018)
17. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
18. Vinyals, O., et al.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)
19. Wang, Z., Wu, W., Huang, Z.: Scalable multi-agent reinforcement learning architecture for semi-MDP real-time strategy games. In: Zhang, H., Zhang, Z., Wu, Z., Hao, T. (eds.) NCAA 2020. CCIS, vol. 1265, pp. 433–446. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-7670-6_36
20. Yang, Z., Ontanón, S.: Guiding Monte Carlo tree search by scripts in real-time strategy games. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, vol. 15, pp. 100–106 (2019)
21. Ye, D., et al.: Towards playing full MOBA games with deep reinforcement learning. In: Advances in Neural Information Processing Systems 33, pp. 621–632 (2020)