8th International Conference on Information Science and Technology
June 30-July 6, 2018; Granada, Cordoba, and Seville, Spain

Reinforcement Learning for Build-Order Production in StarCraft II

Zhentao Tang, Dongbin Zhao, and Yuanheng Zhu
The State Key Laboratory of Management and Control for Complex Systems,
Institute of Automation, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China 100190
Email: {tangzhentao2016, dongbin.zhao, yuanheng.zhu}@ia.ac.cn

Ping Guo
School of Systems Science
Beijing Normal University
Beijing, China 100875
Email: pguo@bnu.edu.cn

***Abstract*—StarCraft II is one of the most popular real-time strategy games and has become an important benchmark for AI research as it provides a complex environment with numerous challenges. The build order problem is one of the key challenges which concern the order and type of buildings and units to produce based on current game situation. In contrast to existing hand-craft methods, we propose two reinforcement learning based models: Neural Network Fitted Q-Learning (NNFQ) and Convolutional Neural Network Fitted Q-Learning (CNNFQ). NNFQ and CNNFQ have been applied into a simple bot for fighting against the enemy race. Experimental results show that both these two models are capable of finding the most effective production sequence to defeat the opponent.**

***Keywords*—reinforcement learning; deep learning; build-order; production; StarCraft II**

I. Introduction

Reinforcement learning has been proved as a useful framework for training an agent to take most suitable action in one environment. Games have been considered as an excellent scene for testing the capability of agents performance after training by reinforcement learning. In recent years, many games have been overcome by deep learning and reinforcement learning methods [1], such as Atari [2], AlphaGo [3], AlphaGo Zero [4], DeepStack [5], VizDoom [6], Alpha Zero for Chess and Shogi [7], self-playing learning for gomoku [8], [9], battle units micro management [10], Convolutional fitted Q iteration for vision-based control [11]. Games artificial intelligent (AI) becomes more and more popular research area now.

Nevertheless, Real-Time Strategy (RTS) games are still an open challenge for currently reinforcement learning method. RTS games are highly complex on several levels for their enormous states and action spaces with a large number of units which have to be controlled in real-time. StarCraft is a masterwork of RTS Game. The game mainly consists of two players manage to defeat each other over a square-shaped map. Each player should spawn workers and construct the buildings for producing troops by available resources. It is called macro management which is one of the keys to victory in StarCraft. Producing the suitable unit according to the scene should be considered as the first key step to create RTS Game AI bot, for different kinds of units have different build-order functions in strategies.

Up to now, bots participating in these competitions, such as Artificial Intelligence and Interactive Digital Entertainment (AIIDE) StarCraft AI Competition, CIG StarCraft Competition, Student StarCraft AI Competition, mainly depend on hard-code rule and tree search methods [12], [13]. The most common method to adapt to different opponents strategies is Upper Confident Bounds(UCB) [14], which still shows its limit of flexibility in different games. Though there are many different opening production orders in one bot, the bot only selects one of the orders at the beginning of the game relying on the history of fighting against the opponent.

To break this limitation, we focus on applying the neural network with reinforcement learning method to macro management tasks in StarCraft II in the context of deciding which unit should be produced next. A neural network is trained to predict these decisions based on reinforcement learning. Then, the bot used a simple micro management method to beat the opponent. Though our approach does not achieve the state-of-the-art results, it is a promising first step towards reinforcement learning methods for macro management in RTS games.

The paper is organized as follows. In Section II, we introduce the background and related works of StarCraft II. Next introduces two approaches NNFQ and CNNFQ for build-order production in StarCraft II. The related experiments' results are shown in Section V. Our conclusion and future work is summarized in the end.

II. Background and Relate Work

StarCraft II is a real-time strategy video game released by Blizzard in 2010 and it is a sequel to the game StarCraft. The game revolves around three races: Terran, Zerg, and Protoss. The player begins to select one of these three races. Then, workers must gather resources to produce battle units, then player command these battle units to fight against the opponent, and the player will win victory when the battle units wipe out the opponent's all building units. The screenshot shows that workers of Zerg are collecting minerals and gases for producing battle units.

Google DeepMind has already put their effort into the research of StarCraft II. But creating a totally end-to-end Bot for StarCraft by deep reinforcement learning seems an

Fig. 1. The screen shot of StarCraft II in Zerg player's vision.

impossible challenge, they released the platform (PySC2) [15] with Blizzards API to the public and this has encouraged more and more researchers to put together to realize this goal. PySC2 is Python-based component of the StarCraft II Learning Environment (SC2LE). In addition, PySC2 not only provides the API in Python wrapper, but also designed some maps for reinforcement research, including Melee, Ladder, and Mini Games. Here we take Melee maps as the test bed for macro management. Melee maps are made specifically for machine learning, and they resemble Ladder maps but should be smaller. These are mainly two kinds of maps in Melee, one is called Flat maps, which have no special terrain and easy for attacking the opponent. The other is called Simple maps, which are more normal with base-expansions, ramps, and lanes of attack, but still are smaller than normal ladder maps.

Here we introduce some existing approaches for dealing with the build-order problems in RTS games. These methods can be boiled down into the case-based method, search method, evolutionary algorithm, Bayesian model, replay-based method. Case-based reasoning system [16] was designed for selecting build orders in a real-time strategy game in early years. The case retrieval process generalizes features of the game state and chooses cases using specific recall methods, which matches with a subset of the case features. Churchill et al. [17] considered the build order problem in StarCraft as to find suitable action sequences, which was constrained by resources and unit dependencies, and utilized a depth-first branch and bound algorithm, in the shortest possible time span. Build-order planning problem has been disposed of multi-object evolutionary algorithm [18], [19] by encoding the list of builds as genotype. Nevertheless, these approaches above were only designed to find an opening build-order and do not take the opponent's strategy into account with the time passed. Bayesian model [20] was put forward to predict the opponent's strategy from imperfect information obtained by a scout. However, this method heavily relies on hard-coded engineering on the top of the prediction model. Replay-based methods [21]–[23] regard replays as the datasets for supervised learning. Since there are large enough amount of replays [24], using deep learning to learn macro management in StarCraft from replays [25] becomes practical.

However, these methods mentioned above have respective shortcomings. The case-based system requires large enough amount of hard-coded rules. Search method should take enough time to ensure the searching precision of prediction for suitable build-order. The evolutionary algorithm needs abundant computing resources when the search space is large. Replay-based methods depend on the reliability of data. And it is really a heavy work for the researcher to ensure the quality of each replay data. In contrast to these methods, we propose two approaches for agent learning to make a suitable decision in building order by Neural Network Fitted Q-learning (NNFQ) and Convolutional Neural Network Fitted Q-learning (CNNFQ).

III. Neural Network Fitted Q-Learning

We train a 4-layer fully connected neural network to select human macro management actions, and decide what to produce in a given state. The macro management control center seems a single agent in neural network fitted Q-Learning framework, which generates its decision based on the state s in every step.

A. Input and Output Parts for NNFQ

In this method, we define the entire state space vectors of input which consists of a few sub-vectors described here in order, in which the numbers represent the indexes in the input vectors. Since StarCraft II is an exactly complicated RTS game for game AI research. To simplify the problem, Zerg is only selected as the object of training for macro management.

1) **0:** The time passed from the beginning.
2) **1-4:** Enemy race, including Terran, Zerg, Protoss, and Random.
3) **5-6:** The number of my supply has been occupied and the number of mineral and Vespene geyser areas on map.
4) **7-14:** The number of my all types of producing buildings and technology buildings.
5) **15-23:** The number of enemy's all producing buildings and technology buildings for Zerg.
6) **24-35:** The number of enemy's all producing buildings and technology buildings for Terran.
7) **36-47:** The number of enemy's all producing buildings and technology buildings for Protoss.
8) **48-53:** The number of my all types of combat units.
9) **54-59:** The number of enemy's all type of combat units for Zerg.
10) **60-66:** The number of enemy's all type of combat units for Terran.
11) **67-75:** The number of enemy's all type of combat units for Protoss.
12) **76-83:** Economy state, including mineral, gas, the number of us and enemy's extension bases and workers.

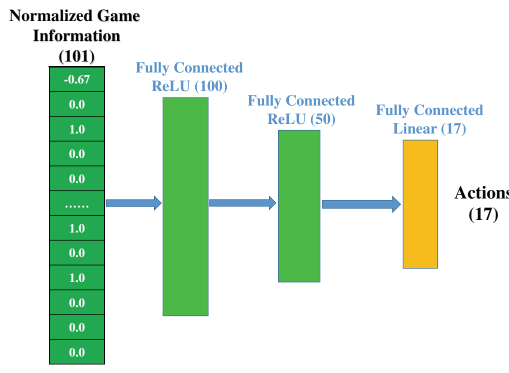
Normalized Game Information (101)
-0.67
0.0
1.0
0.0
0.0
......
1.0
0.0
1.0
0.0
0.0
0.0
Fully Connected ReLU (100)
Fully Connected ReLU (50)
Fully Connected Linear (17)
Actions (17)

Fig. 2. Architecture of NNFQ.

13) **84-100:** Last execution of high level actions, including no ops.

Here is totally 101 feature values as a basic representation of game information. These feature values are all extracted from the raw game information through StarCraft II API. All input values are normalized into the interval [-1, 1]. The number of units is normalized by the maximum of type. The normalized form is:

$$value_{norm} = \frac{value_{cur} - value_{max}}{value_{max}}, \tag{1}$$

where $value_{cur}$ and $value_{max}$ are the current value and max value of feature, which are obtained by the API. Besides, we define the output of agent's actions, and these actions should be divided into two forms, unit production, and building production. And there are totally 17 high-level actions, such as training drone, overlord, zergling, queen, roach and building hatchery, extractor, spawning pool, roach warren, etc.

B. Network Architecture of NNFQ

Since the input vectors are one-dimensional data, a simple multi-layered network architecture with fully-connected layers is used. The input layer has 101 units, which are based on the game state definition we made before. The first hidden layer of 100 units and the second layer of 50 used the same ReLU (Rectified Linear Units) activation function. The output layer has 17 neurons with linear activation function, and it means Zerg player can directly decide which type of unit or building should be produced as next action by the Q values. In other words, the Q values reflect which actions will more suitable for current state.

C. Training in NNFQ

NNFQ model uses Double Q Network [26] and Prioritized Experience Replay [27] as the training tricks. Temporal-Difference(TD) n-step learning is considered as a useful way for training, and we use TD(n) rather than TD(0) as a backup for it is able to roughly obtain a less biased estimation in an off-policy way, such Q-Learning. Noticed that each action made by NNFQ is recorded as the training data into the memory pool. Here we set fixed learning rate $lr = 0.0013$, TD step $n = 10$, replay memory size $m_{size} = 10000$, discount rate $\gamma = 0.99$, training batch size is 16, exploration rate is decreased linearly from 0.8 to 0.0 in 1,000 matches. Target network's weights are updated from the training network every 10 matches, and RMSProp algorithm is used to optimize the loss function. Related forms of the train are listed.

$$r_i = \begin{cases} 10, \textit{if the game ends with a win,} \\ -10, \textit{if the game ends with a loss,} \\ 0, \textit{if step } i \textit{ is not final step.} \end{cases} \tag{2}$$

$$R_t = \sum_{i=t}^{t+n} \gamma^{i-t} r_i, \tag{3}$$

$$Q(s,a) = \begin{cases} R_t + \gamma Q^{target}(s', \arg\max_{a'} Q^{train}(s',a')), \\ \qquad \textit{if state } s \textit{ is not terminal,} \\ R_t, \textit{if state } s \textit{ is terminal.} \end{cases} \tag{4}$$

r_i is the immediate reward for step i. R_t is the cumulative reward corresponding to step t. Q^{target} and Q^{train} are target Q network and train Q network respectively. And $Q(s,a)$ is the typical Q value function. Beyond that the batch mean square loss function is

$$l = \frac{1}{2} \sum_{(s,a)\in(S,A)} (Q(s,a) - \max_{a'} Q^{train}(s,a'))^2, \tag{5}$$

(S,A) is the training batch. Selecting batch training datas depends on the last Bellman error $|\delta|$. In other words, the larger error $|\delta|$ is made by the data means that the data is more likely to be chosen into the training batch:

$$|\delta| = |Q(s,a) - \max_{a'} Q^{train}(s,a')|. \tag{6}$$

IV. Convolutional Neural Network Fitted Q-Learning

Convolutional Neural Network fitted Q-learning is based on the neural network fitted Q-Learning. In contrast to the NNFQ, CNNFQ is added with convolutional neural network layers as another input-hidden-connected part. Besides, we make some differences in the output layer, and the network architecture looks like [28], which is called dueling architecture.

A. Input and Output Parts for CNNFQ

Since the CNNFQ is an extension from NNFQ, CNNFQ's and NNFQ's one-dimensional-data input part are identical. But we take account of different kinds of game state information, what is called situation assessment. Due to the Blizzard's API, we are able to obtain our and enemy's units information when they in our vision, especially units location information. The map is divided into a 64×64 square chess-board. Like chesses on the board, we put all units on the board according to the unit's location. Though unit's location (x,y) is a real number, it should be rounded as integer number in [0, 63]. Hence, we obtain two planes, one for my units and the other one for

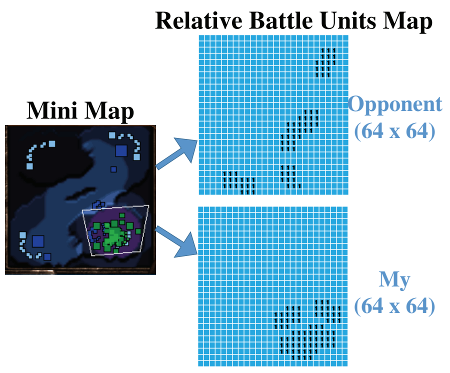

Fig. 3. Conversion from Mini Map to Relative Map.

opponent's units. In order to reinforce the input plane, every unit is considered as a 3×3 square and it's integer numeric location is the center of the 3×3 square.

CNNFQ's output part is another difference from the NNFQ. Owe to situation assessment by these two input planes, we access to estimate the final win rate by setting up an output node in the output part.

B. Network Architecture of CNNFQ

CNNFQ's input part has some difference from NNFQ. Though numeric input part keeps the same with 101 neurons as before, we add three convolutional neural network layers with each corresponding max-pooling layer into the network. Each convolutional kernel's size is 3×3 with ReLU activation function, and max-pooling layer's size is also 3×3. In the middle part of network architecture, we put these two parts together. Using a fully-connected layer links these two parts, where has 80 neurons. Ultimately, the output layer is divided into two parts. One part is called the actions, which is the same as NNFQ. The other part is called win rate, which only has one output node with tanh activation function. Currently, the win rate output is only used in training stage, but not in test stage. Nevertheless, we are managing to serve it as the guidance for selecting the suitable strategy of attack.

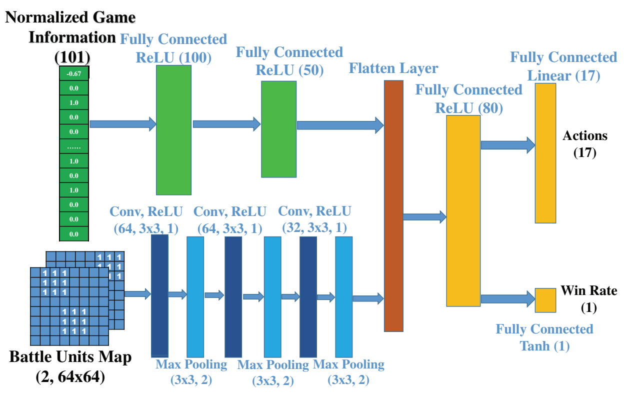

Fig. 4. Architecture of CNNFQ.

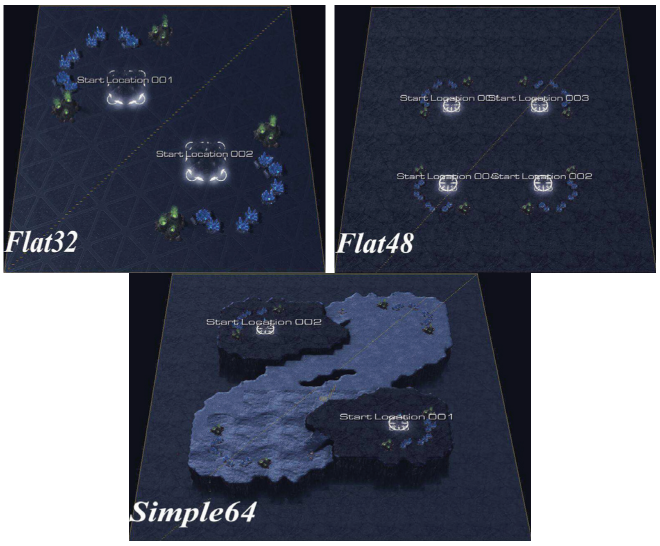

Fig. 5. Melee maps of StarCraft II.

C. Training in CNNFQ

CNNFQ model uses the same training tricks as NNFQ, and most training parameters are kept the same as NNFQ's. Nevertheless, CNNFQ's architecture is different to the NNFQ's, especially the output part of CNNFQ. To train the prediction of win rate, we define the win reward function which reflects the situation in game,

$$r = \sum_{x \in n_1, y \in n_2} 0.05(x_{supplyCost} - y_{supplyCost}), \tag{7}$$

$$r_{winRate} = clip(r, -0.99, 0.99), \tag{8}$$

where n_1 and n_2 denote my battle units and opponent's battle units respectively, and the corresponding $x_{supplyCost}$ is the supply cost of the unit. In other words, $x_{supplyCost}$ reflects the strength of unit. (7) is normalized by a certain coefficient 0.05 for value ranging between [-0.99, 0.99]. Function clip truncates reward $r_{winRate}$ into a certain range in case of overflow. In addition, there is some difference in the loss function. The loss function consists of two parts, one is $loss_Q$, the other is $loss_V$. Here we describe that

$$loss_Q = \frac{1}{2} \sum_{(s,a) \in (S,A)} (Q(s,a) - \max_{a'} Q^{train}(s,a'))^2, \tag{9}$$

$$loss_V = \frac{1}{2} \sum_{s \in S_{pos}} (r_{winRate} - V(s))^2, \tag{10}$$

$$loss_{total} = loss_Q + loss_V, \tag{11}$$

here $V(s)$ is the value function, which is from win rate output. And we use the optimizer Adam to minimize the total loss function.

V. Experiments

Google DeepMind provides a suitable test platform for StarCraft II. In our work, we select some of the melee maps as the test bed, which are Flat32, Flat48 and Simple64. Flat32 and Flat48 are the same terrains as flat ground, without any ramp or obstacle, but the size of Flat48 is a little larger than Flat32. As in Fig. 5, there are 4 start locations for Flat48, while there are 2 start locations for Flat32. Simple64 is more complicated in melee maps. In contrast to Flat maps, there are high lands, ramps, and obstacles in Simple64.

In this paper, our reinforcement-learning-based bot is based on Zerg race. Here we apply the NNFQ and CNNFQ as the macro management module into a simple bot, and the simple bot is capable of managing battle units for attacking the opponent's base. Nevertheless, there is not complicated micro management of battle units in the bot, and only the simple bot attacks the enemy when the number of battle units reached the threshold, and the threshold is fixed at 12.

StarCraft II has Build-in scripted AI, and it has 10 levels of different difficulty which are named: very easy(1), easy(2), medium(3), medium hard(4), hard(5), hard very hard(6), very hard(7), cheat vision(8), cheat money(9), cheat insane(10). Noticed that cheat vision, cheat money, and cheat insane are inequality for us. Since our simple bot is lacking completed micro management for those situations, thus the bot can not compete with the scripted bot in those levels. In other words, the reinforcement-learning-based bot is too difficult to improve itself by fighting against built-in AI with cheating level, because it is hard for an agent to adjust itself without any win reward signal.

In our work, we select the most complicated map, Simple64, as the training map. Then, we applied NNFQ and CNNFQ model into the bot fighting against Zerg, Terran, and Protoss respectively. Trained progress is shown in Fig. 6,7. There exists 3 mainly and obviously differences in performances of these two models. First, CNNFQ converges quickly to a stable value, while NNFQ does not. Secondly, CNNFQ finds out the effective build-order arrangement for defeating opponent earlier than NNFQ. What we can see from Figs. 6,7, CNNFQ just runs around 150 epochs for reaching the high win rate against all three races, while NNFQ requires nearly 1000 epochs for achieving the same goal. Last but not least, CNNFQ's learning curve of win rate is rising steadily in comparison with NNFQ's.

For the test, we use NNFQ's and CNNFQ's models as the testing benchmark, while their opponents are the same built-in AI bot with just six different difficulty levels. Since the very easy level is too easy for bot achieving 100% win rate with fighting against all 3 races in 40 games, the result of very easy level is not listed here. After test, results are listed in Table I-III.

These tables show us that NNFQ and CNNFQ both perform best versus Terran, even CNNFQ has achieved 100% win rate whatever the map and the difficulty level. Nevertheless, NNFQ and CNNFQ perform slightly poor when the opponent is Zerg or Protoss in Simple64.

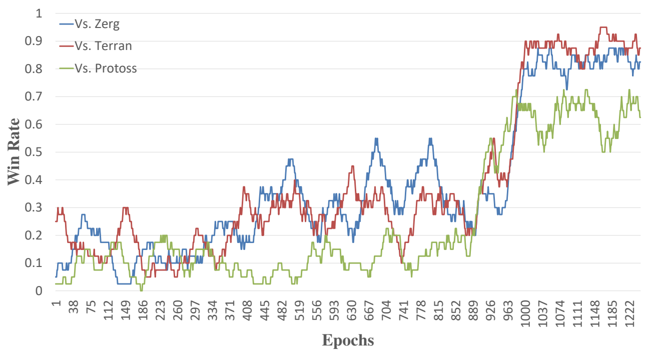

Fig. 6. NNFQ's win rate against 3 Races in Simple64.

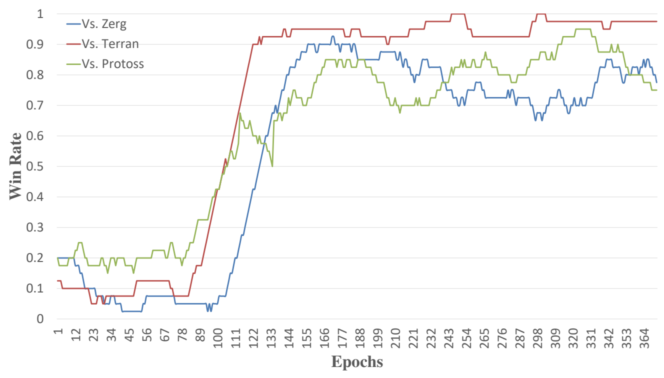

Fig. 7. CNNFQ's win rate against 3 Races in Simple64.

Experiments show that some loss is not caused by inappropriate build-order arrangement, but is caused by lacking appropriate micro management of battle units or suitable strategy of attack. In spite of this, NNFQ and CNNFQ models have proved their capability of acquiring reasonable and suitable build-order action defeating the opponent through deep reinforcement learning.

VI. Conclusion and Future Work

In this paper, we propose two reinforcement-learning-based models are called NNFQ and CNNFQ. NNFQ only makes most of the numeric information from the game for deciding which unit should be built out, while CNNFQ, the improved version of NNFQ, even utilizes the unit's location information and adds a separated output node is called situation assess-

TABLE I
NNFQ & CNNFQ vs Zerg (in 40 games)

Model	Maps	Win Rate against Different Levels(%)					
		2	3	4	5	6	7
NNFQ	Flat32	100	100	85	100	95	100
	Flat48	95	90	100	90	95	100
	Simple64	90	85	80	80	80	75
CNNFQ	Flat32	100	100	100	100	100	100
	Flat48	100	100	100	100	100	100
	Simple64	90	87.5	82.5	82.5	87.5	80

TABLE II
NNFQ & CNNFQ VS TERRAN (IN 40 GAMES)

Model	Maps	Win Rate against Difficulty Levels(%)					
		2	3	4	5	6	7
NNFQ	Flat32	100	100	100	100	100	100
	Flat48	100	100	100	95	100	100
	Simple64	100	95	92.5	95	90	90
CNNFQ	Flat32	100	100	100	100	100	100
	Flat48	100	100	100	100	100	100
	Simple64	100	100	100	100	100	100

TABLE III
NNFQ & CNNFQ VS PROTOSS (IN 40 GAMES)

Model	Maps	Win Rate against Different Levels(%)					
		2	3	4	5	6	7
NNFQ	Flat32	100	100	100	100	100	100
	Flat48	100	100	95	100	100	95
	Simple64	90	82.5	82.5	75	72.5	75
CNNFQ	Flat32	100	100	100	100	100	100
	Flat48	100	100	100	100	100	100
	Simple64	100	85	85	80	80	82.5

ment, which can decide when should our battle units attack the enemy base. They both have shown their abilities to learn to obtain some rational build-order arrangements according to the real-time situation in the game, in particular, finding out the most effective production sequence for defeating the opponent, such as Zerglings Rush, Roaches Rush, even Hydralisks Rush. Comparing with NNFQ, CNNFQ model is a more efficient method for acquisition of the valid and rational build-order management. However, mastering the game of RTS is an extremely difficult task in artificial intelligence, because of the complexity in such games. An excellent bot in StarCraft not only requires performing well in macro management but also requires ingenious micro management to cooperate with the macro management.

Our future research will focus on the cooperation of the macro management and micro management for more complicated terrain and scene by other deep learning and reinforcement learning algorithms. Our goal is to narrow the gap between humans and computers in such RTS-games like Starcraft II.

ACKNOWLEDGEMENT

This work is supported by National Natural Science Foundation of China (NSFC) under Grants No. 61573353 and No. 61533017. Thanks for the open source codes from Dave Churchill's Command Center and Sijia Xu's Overkill. Their codes save us much time in learning how to use the platform of StarCraft II and make a game-AI bot.

REFERENCES

[1] Z. Tang, K. Shao, D. Zhao, et al, "Recent progress of deep reinforcement learning: from AlphaGo to AlphaGo Zero," *Control Theory & Applications,* vol. 34, no. 12, pp. 1529-1546, 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, et al, "Playing atari with deep reinforcement learning," arXiv preprint arXiv:1312.5602, 2013.

[3] D. Silver, A. Huang, C. J. Maddison, et al, "Mastering the game of Go with deep neural networks and tree search," *Nature,* vol. 529, no. 7587, pp. 484-489, 2016.

[4] D. Silver, J. Schrittwieser, K. Simonyan, et al, "Mastering the game of go without human knowledge," *Nature,* vol. 550, no. 7676, pp. 354-359, 2017.

[5] M. Moravčík, M. Schmid, N. Burch, et al, "DeepStack: Expert-level artificial intelligence in heads-up no-limit poker," Science, vol. 356, no. 6337, pp. 508-513, 2017.

[6] G. Lample and D. S. Chaplot, "Playing FPS Games with deep reinforcement learning," AAAI Conference on Artificial Intelligence, pp. 2140-2146, 2017.

[7] D. Silver, T. Hubert, J. Schrittwieser, et al, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," arXiv:1712.01815, 2017.

[8] D. Zhao, Z. Zhang, and Y. Dai, "Self-teaching adaptive dynamic programming for Gomoku," *Neurocomputing,* vol. 78, pp. 23-29, 2012.

[9] Z. Tang, D. Zhao, K. Shao, et al, "ADP with MCTS algorithm for Gomoku," *IEEE Symposium Series on Computational Intelligence,* pp. 1-7, 2016.

[10] K. Shao, Y. Zhu, and D. Zhao, "Cooperative reinforcement learning for multiple units combat in StarCraft," *IEEE Symposium Series on Computational Intelligence,* pp. 1-6, 2017.

[11] D. Zhao, Y. Zhu, L. Lv, et al, "Convolutional fitted Q iteration for vision-based control problems," *International Joint Conference on Neural Network,* pp. 4539-4544, 2016

[12] S. Ontanon, G. Synnaeve, A. Uriarte, et al, "A survey of real-time strategy game AI research and competition in StarCraft," *IEEE Transactions on Computational Intelligence & Ai in Games,* vol. 5, no. 4, pp. 293-311, 2013.

[13] D. Churchill, M. Preuss, F. Richoux, et al, "StarCraft bots and competitions," *Encyclopedia of Computer Graphics and Games,* pp. 1-18, 2016.

[14] C. B. Browne, E. Powley, D. Whitehouse, et al, "A Survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence & Ai in Games,* vol. 4, no. 1, pp. 1-43, 2012.

[15] O. Vinyals, T. Ewalds, S. Bartunov, et al, "StarCraft II: A new challenge for reinforcement learning," arXiv:1708.04782, 2017.

[16] B. G. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 106-111, 2009.

[17] D. Churchill and M. Buro, "Build order optimization in StarCraft," AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, pp. 14-19, 2011.

[18] H. Köstler and B. Gmeiner, "A multi-objective genetic algorithm for build order optimization in StarCraft II," *KI - Künstliche Intelligenz,* vol. 27, no. 3, pp. 221-233, 2013.

[19] N. Justesen and S. Risi, "Continual online evolutionary planning for in-game build order adaptation in StarCraft," the Genetic and Evolutionary Computation Conference, pp. 187-194, 2017

[20] G. Synnaeve and P. Bessiere, "A bayesian model for plan recognition in RTS games applied to StarCraft," AAAI Conference on Artificial Intelligence and Interactive Digital Entertainmen, vol. 33, no. 203, pp. 79-84, 2011.

[21] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," *IEEE Symposium on Computational Intelligence and Games,* pp. 140-147, 2009.

[22] A. A. B. Branquinho and C. R. Lopes, "Planning for resource production in real-time strategy games based on partial order planning, search and learning," *IEEE International Conference on Systems Man and Cybernetics,* pp. 4205-4211, 2010.

[23] H. C. Cho, K. J. Kim, and S. B. Cho, "Replay-based strategy prediction and build order adaptation for StarCraft AI bots," *IEEE Conference on Computational Intelligence in Games,* pp. 1-7, 2013.

[24] Z. Lin, J. Gehring, V. Khalidov, et al, "STARDATA: A StarCraft AI research dataset," arXiv: 1708.02139, 2017.

[25] N. Justesen and S. Risi, "Learning macromanagement in StarCraft from replays using deep learning," IEEE Conference on Computational Intelligence and Games, pp. 162-169, 2017.

[26] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," AAAI Conference on Artificial Intelligence, pp. 2094-2100, 2016.

[27] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," arXiv:1511.05952, 2015.

[28] Z. Wang, T. Schaul, M. Hessel, et al, "Dueling network architectures for deep reinforcement learning," arXiv:1511.06581, 2015.