Proceedings of the Fifteenth AAAI Conference on Artificial
Intelligence and Interactive Digital Entertainment (AIIDE-19)

Macro Action Selection with Deep Reinforcement Learning in StarCraft

Sijia Xu,[1] Hongyu Kuang,[2] Zhuang Zhi,[1] Renjie Hu,[1] Yang Liu,[1] Huyang Sun[1]
[1]Bilibili
[2]State Key Lab for Novel Software Technology, Nanjing University
{xusijia, zhuangzhi, hurenjie, liuyang01, sunhuyang}@bilibili.com, khy@nju.edu.cn

Abstract

StarCraft (SC) is one of the most popular and successful Real Time Strategy (RTS) games. In recent years, SC is also widely accepted as a challenging testbed for AI research because of its enormous state space, partially observed information, multi-agent collaboration, and so on. With the help of annual AIIDE and CIG competitions, a growing number of SC bots are proposed and continuously improved. However, a large gap remains between the top-level bot and the professional human player. One vital reason is that current SC bots mainly rely on predefined rules to select macro actions during their games. These rules are not scalable and efficient enough to cope with the enormous yet partially observed state space in the game. In this paper, we propose a deep reinforcement learning (DRL) framework to improve the selection of macro actions. Our framework is based on the combination of the Ape-X DQN and the Long-Short-Term-Memory (LSTM). We use this framework to build our bot, named as LastOrder. Our evaluation, based on training against all bots from the AIIDE 2017 StarCraft AI competition set, shows that LastOrder achieves an 83% winning rate, outperforming 26 bots in total 28 entrants.

1 Introduction

StarCraft: Brood War (SC) is one of the most popular and successful Real Time Strategy (RTS) games created by Blizzard Entertainment in 1998. Under the setting of a science-fiction based universe, the player of SC picks up one of the three races: Terran, Protoss, or Zerg, to defeat other players in a chosen map. Figure 1 presents a screenshot of SC showing a play using the Zerg race. In general, to achieve victory in a standard SC game, the player needs to perform a variety of actions, including gathering resources, producing units, updating technologies and attacking enemy units. These actions can be categorized into two basic types: micro actions (Micro) and macro actions (Macro) (Ontanón et al. 2013):

Micro. The micro actions manipulate units to perform operation-level tasks such as exploring regions in a map, collecting resources and attacking the enemy units. The general goals of micro actions during the entire game are: (1) to keep units performing more tasks; (2) to avoid being eliminated by the enemy.

Figure 1: A screenshot of StarCraft: Brood War

Macro. The macro actions represent the strategy-level planning to compete with the opponent in the game, such as the production of combat units, the placement of different buildings and the decision of an attack. The general goal of macro actions is to efficiently counter the opponent's macro actions throughout the game. It is worth noting that, in a standard SC game, regions in the map that are not occupied by the player's units or buildings are kept unknown. The so-called "fog of war" mechanism leads to partial observations on the map and thus significantly increases the difficulty to select proper macro actions.

In recent years, SC has been widely accepted as a challenging testbed for AI researchers, because of the multi-unit collaboration in the Micro and the decision-making on enormous state spaces in the Macro. A growing number of bots based on different AI techniques are proposed and continuously improved, especially in annual StarCraft AI competitions held by both AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), and IEEE Conference on Computational Intelligence and Games (CIG). Unfortunately, although the ability of SC bots has been greatly improved, a large gap remains between the top-level bot and the professional human player. For example, in the game ladder iCCup where players and bots are ranked by their SC match results (Ontanón et al. 2013), the best SC bot ranks from D to D+. On the contrary, the average amateur player ranks from C+ to B, while the professional player usually ranks from A- to A+. One vital reason why bots fall behind human players is that current bots mainly rely on predefined rules to perform macro actions. These rules are not scalable and efficient enough to cope with the enormous yet partially observed state space in the game.

Recently, DRL-based bots have achieved substantial progress in a wide range of games, such as Atari games (Mnih et al. 2015), Go (Silver et al. 2017), Doom (Wu and Tian 2016) and Dota2 (OpenAI 2018). However, two main challenges remain when using the DRL framework to perform better macro actions in SC. The first one is the partial observation problem caused by the fog of war in the map. This problem makes the current observations insufficient to infer the future states and rewards. The other challenge is the sparse reward problem, for example, in SC the length of a usual game is around 20 minutes with 250-400 macro actions performed, thus it is hard to get positive rewards using only the terminal reward at the end of the game when training from scratch.

To address these two challenges, we combine the reinforcement learning approach Ape-X DQN (Horgan et al. 2018) with the Long-Short-Term-Memory model (LSTM) (Hochreiter and Schmidhuber 1997) to propose our DRL-based framework. Specifically, the LSTM is used to address the partial observation problem, and the Ape-X is used to address the sparse reward problem. Then, we use this framework to build our bot, named as LastOrder[1]. By training against all bots from the AIIDE 2017 StarCraft AI Competition, LastOrder achieves an overall 83% win-rate, outperforming 26 bots in total 28 entrants. Furthermore, the same version of LastOrder attends the AIIDE 2018 StarCraft Competition and ranks 11 in total 25 entrants. We have also open-sourced LastOrder with the training framework at https://github.com/Bilibili/LastOrder.

2 Related Work

Researchers have focused on macro action selection in StarCraft for a long time (Ontanón et al. 2013). One way used by many bots, e.g., UAlbertaBot (Churchill 2018b) and AIUR (AIUR 2018), is the bandit learning method based on several predefined macro action sequences. Bandit learning can choose the most appropriate macro action sequence using the historical match. However, In SC, typical bandit learning method is only used at the start of the game. Thus, it's insufficient to cope with the large state space throughout the game.

Meanwhile, a different way to solve this problem is the data mining from human players' replays. Hsieh et al. propose to learn actions from a large number of replays (Hsieh and Sun 2008). They learn every detailed click from human players. Weber et al. encode the whole game into a vector for each player and then model the problem into a supervised learning problem (Weber and Mateas 2009). Kim et al. propose to categorize different building orders and summary them into separate states and actions (Kim et al. 2010). Different from the above work, Hostetler et al. start to consider partially observe problem in RTS game by using dynamic Bayes model inference (Hostetler et al. 2012). Many researchers (Synnaeve, Bessiere, and others 2012) (Robertson and Watson 2014) also propose datasets of replays which contain much richer information. However, for full bot development, replay mining method remains unsatisfying for two reasons. First, the macro action sequence from professional players may not be the best choice for the bots, due to the large difference in micro management ability between bots and professional players. Second, some tactic-level macro actions, e.g., when to trigger a specific kind of attack to a destination, are important to the overall performance of bots. But it is unlikely for replay mining method to extract these highly abstracted tactic-level actions.

In recent years, DRL-based methods have achieved noticeable success in building autonomous agents. Currently, there are mainly two ways to apply DRL to RTS games. The first one is applying DRL to the micro management. Kong et al. (Kong et al. 2017) (Kong et al.) propose several master-slave and multi-agent models to help controlling each unit separately in specific scenario. Shao et al. (Shao, Zhu, and Zhao 2018) introduce transfer learning and sharing multi-agent gradient to train cooperative behaviors between units. These proposed methods are capable of performing excellent work in a small combat scenario, but it remains hard to scale to a large combat scenario and to react instantly at the same time. This limitation also restricts the practical application of DRL based micro management in full bot development. The other way is applying DRL to the macro management. Compared with replay-mining methods, macro actions learned through DRL can directly match the bot's micro management ability. Furthermore, macro actions learned from DRL can include both native macro actions (e.g., building, producing, and upgrading) and customized tactic-level macro actions. Sun et al. (Sun et al. 2018) created a StarCraft II bot based on a DRL framework to do macro actions selection, and achieved desirable results when competing with the build-in Zerg AI. By contrast, we focus more on handling partial observation and sparse reward problems to compete against a wider range of bots.

3 Proposed Method

3.1 Actions, state and reward

Macro actions We define 54 macro actions for Zerg covering the production, building, upgrading and different kinds of attack as summarized in Table 1(for the full list please refer to our project on GitHub). All Macro actions excluding the attack actions have direct meaning in StarCraft. For attack actions, each one represents a attack with a specific mode (e.g., harassing enemy natural base using mutalisks).

Micro actions We define a large set of detailed micro actions to manipulate different units and buildings of the Zerg race. In general, these actions manipulate units and buildings to perform operation-level tasks, such as moving to somewhere and attacking other units.

[1]LastOrder is the name of an immature AI character in a popular animated drama called Toaru Majutsu no Index

Table 1: Summary of 54 macro actions

Category	Actions	Examples
Production	7	ProduceZergling
Building	14	BuildLair
Upgrading	15	UpgradeZerglingsSpeed
Attack	17	AttackMutaliskHarassNatural
Expansion	1	ExpandBase
Waiting	1	WaitDoNothing

State The state come as a set of current observation features and history features. Current observation features describe the current status of ours and enemies, e.g., units and buildings. History features are mainly designed to keep and accumulate enemy information from the start of the game. For example, once we observe a new enemy building, unless it is destroyed by us, we add a feature describing its existence whether the enemy building is under the fog of war.

Reward Reward shaping is an effective technique to reinforcement learning in a complicated environment with delayed reward (Ng, Harada, and Russell 1999). In our case, instead of using a terminal result (1(win)/-1(loss)), we use a modified terminal reward with in-game score as below (where *timedecay* refers to the game time):

$$r_{new} = \gamma^{timedecay} \frac{score_{our} - score_{enemy}}{\max\left(score_{our}, score_{enemy}\right)} \quad (1)$$

The in-game score is defined by the SC game engine including the a building score, a unit score and a resource score to reflect the player's overall performance. The modified terminal score describes the quality of the terminal result. We find that it can guide the exploration more efficiently. For example, policy in a game with a higher modified terminal score is better than others even though games are all lose.

3.2 Learning Algorithms and Network Architectures

Ape-X DQN In SC, it is hard to get positive reward using only the terminal reward at the end of a game when training from scratch. This sparse reward problem become severe when training against strong opponent. Recently, a scaled up variant DQN called Ape-X DQN (Horgan et al. 2018) achieves a new state of arts performance on Atari games, especially on some well-known sparse reward game like Montezuma's Revenge. They suggest that using a large number of actors may help to discover new courses of policy and avoid the local optimum problem caused by insufficient exploration. This scaled up approach is a relatively easy way to solve the sparse reward problem.

Specifically, Ape-X DQN uses double Q-learning, multi-step bootstrap targets, prioritized replay buffer and duel network. In our case, there is no instant reward, the loss function is $l_t(\theta) = \frac{1}{2}(G_t - q(S_t, A_t, \theta))^2$ with the following definition (where *t* is a time index for the sample from the replay starting with state S_t and action A_t, and θ^- denotes parameters of the target network):

$$G_t = \lambda^n q(S_{t+n}, \underset{a}{argmax}\, q(S_{t+n}, a, \theta), \theta^-) \quad (2)$$

Although it is uncommon to use multi-step bootstrap targets in off-policy learning without off-policy correction, we find that this setting performs better than single step target setting. We suggest that using low exploration rate on majority of actors may improve the on-policy degree of training data, and the performance of multi-step bootstrap targets is also improved in off-policy setting with the on-policy training data. Besides, in our case, instead of using a center replay memory to store all transitions in FIFO order, we split the replay memory into multiple segments which equal to the number of opponents, due to the unbalanced training transition generating speed of different opponents. Each opponent only update transition on its own replay memory segment in FIFO order. During the training stage, transition is sampled over all replay memory segments according to its prioritization.

Deep Recurrent Q-Networks In SC, the macro action selection decision process is non-Markovian, because the existence of fog of war causes the future states and rewards depending on more than current observation. The macro action selection process becomes a Partially Observable Markov decision process (POMDP) and the DQN's performance declines when given incomplete state observations. To deal with this problem, Hausknecht and Stone (Hausknecht and Stone 2015) introduce the Deep Recurrent Q-Networks (DRQN). According to their experiment, adding a LSTM layer to the normal DQN model can approximate more accurate Q-values, leading to better policies in partially observed environments.

In our case, we use the Ape-X DQN instead of the normal DQN to achieve better performance. Besides, in order to cover longer time horizon, the interval between each LSTM step is extended to 10 seconds. We observed that a 10-seconds interval is short enough to reflect changes in macro state without missing too much detail changes.

3.3 Training

We use 1000 machines(actor) to run 500 parallel games against different opponents scheduled by StarCraftAITounrnamentManger (Churchill 2018a). Similar to TorchCraft (Synnaeve et al. 2016), there are two parts in each actor. The model part uses a separate python process to handle macro action selection based on TensorFlow (Abadi et al. 2016), and the other part is a BWAPI (Heinermann 2018) DLL performing all actions in SC. These two parts use a message queue to exchange message between each other.

During each game, we cache observations in the actor's memory and group them into observation sequences. To alleviate the load of leaner which receives transitions, we only send transitions at the end of the game. The generation speed of transition is approximately 20000 per minute. The learner's update speed is about 10 batches (196 batch size) per second. Actors copy the network parameters from the learner every 10 seconds. Each actor $i \in \{0, 1, \ldots, N-1\}$ executes an ϵ_i greedy policy where $\epsilon_i = \epsilon^{1+\frac{i}{N-1}\alpha}$ with

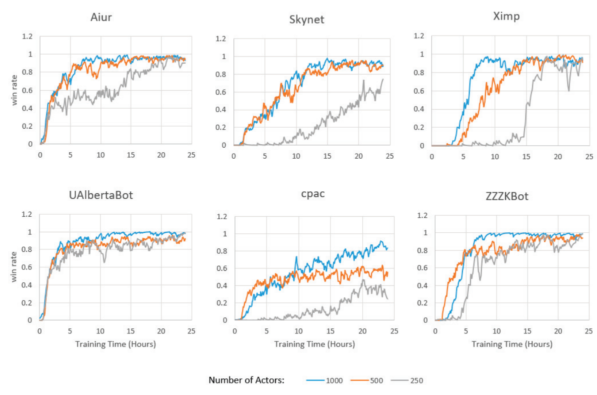

Figure 2: Performance consistently improves as the number of actors increased from 250 to 1000.

Table 2: Six selected opponents (the ranking is based on AI-IDE 2017)

Bot name	Rank	Race	Strategy
ZZZKBot	1	Zerg	Early rush
cpac	4	Zerg	Mid-game
Ximp	13	Protoss	All-in Carrier rush
UAlbertaBot	14	Random	9 strategies on 3 races
Aiur	15	Protoss	4 different strategies
Skynet	17	Protoss	3 different strategies

$\epsilon = 0.4, \alpha = 7, N = 1000$. The capacity of each replay memory segment is limited to one million transitions. Transition is sampled over all replay memory segments according to its proportional prioritization with a priority exponent of 0.6 and an importance sampling exponent set to 0.4.

4 Experiment

4.1 A controlled case for analysis

For qualitative analysis we train our bot against six selected bots as described in Table 2 on a single map. We select Ximp bot and cpac bot to show the sparse reward problem. Both of them use the mid-game rush strategy with stable early game defense and are relatively difficult to defeat. The rest of bots are added to intensify the partial observation problem, because the opponents in the same race usually have some similar macro states during the game.

In Figure 2, we show the learning curve of win-rate against each bot with a different number of actors. The win-rate are evaluated by actors with zero exploration rate. It is worth noting that with 1000 actors almost all the win-rate become close to 1 after 24 hours' training.

A detailed case: LastOrder vs. Ximp We then use Ximp bot for case analysis because the sparse reward problem in Ximp is severest among the six bots. Ximp's early game strategy is to do stable defense with many defense buildings and little combat units. The increasing number of defense buildings and limited combat units are gradually different from other Protoss bots. The mid game strategy of Ximp is to produce top-level combat unit (Carrier) as soon as possible to attack opponent in order to gain big advantage or directly win the game in one shot. Compared with other opponents' counterstrategy, the length of counterstrategy against Ximp is much longer, because LastOrder must first defend Carrier rush in the mid game and then defeat Ximp in the later game. Thus LastOrder needs much more exploration effort in order to get the positive reward when training from scratch.

In Figure 2, after 24 hours' training with 1000 actors, due to the lack of harassing strategy in Ximp, in the early game, LastOrder's counterstrategy is to quickly expand multiple bases and produce workers in order to gain big economy advantage. This is totally different from the learned counter-strategies against other Protoss bots, for example, the strategy against Skynet in the early game is to build defense buildings and to produce combat units in order to counter its early rush.

In the mid game, LastOrder finds the efficient countering battle unit (Scourage) and produce a sufficient number of Scourages to defeat Ximp's Carrier attack, the choice of producing Scourages is also different from counterstrategies of other bots. Due to the big economy advantage in the early game, in the later game LastOrder has sufficient resource to produce other combat units along with the Scourages. Thus it is relatively easy to win the game in the end.

Component analysis We also run additional experiments to improve our understanding of the framework and to investigate the contribution of different components. First, we investigate how the performance scale with the number of actors. We trained our bot with different number of actors (250, 500, 1000) for 24 hours against the 6 opponents set as described in Table 2. Figure 2 shows that the performance consistently improved as the number of actors increased without changing any hyper-parameter or the structure of the network. This is also similar to the conclusion in Ape-X DQN.

Next, In Figure 3, we run three experiments to investigate the influence of other components:

- **Without LSTM.** We only use the latest observation instead of a period of observations to investigate the influence of partial observation. The experiment shows a small performance drop against Protoss bots (Skynet and Ximp). We suggest that model is less likely to differentiate states among the same race without the sequence of observation.
- **Without reward shaping.** We use the origin 1(win)/-1(loss) as the terminal reward instead of the modified terminal reward. According to this experiment, the win-rate of Ximp is kept at about 0% over the whole training time. We suggest that the exploration is much harder to get improved reward without the help of reward shaping to alleviate the sparse reward problem, especially when training against strong opponent.
- **High exploration rate.** In this experiment, each actor executes an ϵ_i greedy policy where $\epsilon_i = \epsilon^{1+\frac{i}{N-1}\alpha}$ with $\epsilon = 0.7, \alpha = 11, N = 1000$. This setting corresponds to a higher exploration rate in actors. The overall performance declines in many bots (cpac, Aiur, Skynet) compared with the low exploration rate setting. We suggest that using low

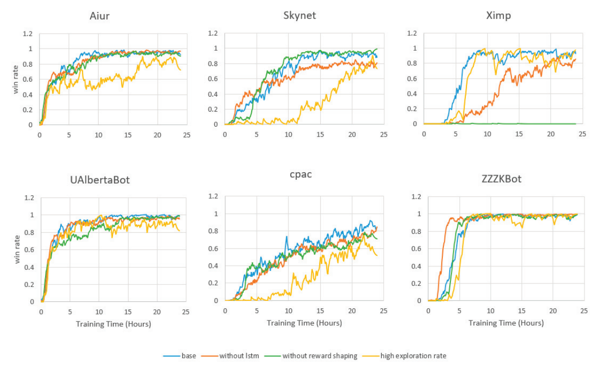

Figure 3: Four different experiments. Each experiment using 1000 actors trained for 24 hours.

exploration rate on majority of actors is equal to getting on-policy transitions with relatively little noisy. As the learning proceeds, the change in policy become less and the transitions in replay buffer become more on-policy. This combination of low exploration on majority of actors and multi-step bootstrap targets improve the overall performance.

4.2 Evaluation on AIIDE 2017 StarCraft AI Competition bots set

We then train our bot against the AIIDE 2017 StarCraft AI Competition bots set over 10 maps to show the performance and scalability. The 2017 bots set is comprised of 28 opponents including 4 Terran bots, 10 Protoss bots and 14 Zerg bots. With the need of parallel training and fast evaluation, we do not use the round robin mode in StarCraftAITounrnamentManger (Churchill 2018a). All games are created as soon as possible without waiting for previous round to finish. This running mode also disable the opponent's online learning strategy and may cause potential performance drop in LastOrder in round robin mode. The final trained LastOrder achieves 83% win-rate in 8000 games, outperforming 26 bots on 28 bots set. The detailed evaluation result is listed in Table 3.

4.3 AIIDE 2018 StarCraft AI Competition

We use the pre-trained LastOrder in 4.2 to attend the AIIDE 2018 StarCraft Competition and rank 11 in total 25 entrants. The official result can be found in (AIIDE 2018).

The performance drop has two reasons. First, when the opponent's micro management suppress ours, our enhancement to macro actions is insufficient to win the game. This happens for SAIDA, CherryPi, Locutus, two Locutus-based bots (CSE and DaQin), and McRave in 2018 competition. Second, The insufficient Terran opponents in the training sets(only 4 Terran bots in the 2017 bots set) leads to low win-rate when playing against new Terran bots Ecgberht and WillyT.

Despite the discussed two reasons, on the rest of newly submitted bots set, LastOrder achieves about 75% win-rate. Considering the use of online learning strategy by other bots in round robin mode, this win-rate is close to the offline evaluation result.

5 Conclusions and Future Work

Developing a strong bot to act properly in StarCraft is a very challenging task. In this paper, we propose a new framework based on DRL to improve the performance of the bot's macro action selection. Via playing against AIIDE 2017 StarCraft Competition bots, our bot achieve 83% win-rate showing promising result.

Our future work involves the following three aspects:

Fix bug and optimize micro. Current macro action execution and micro management is hard coded. The bug in these codes may cause unusual variance in transitions, and it may severely influence training. Besides, the quality of micro is also a key aspect to the performance of bot. e.g., Iron bot used to place building to block the choke point, whereas current LastOrder can not identify the blocked choke point. This usually results in a huge army loss and the macro model cannot help with it.

Self-play training. We observe that to some extent the performance of model is highly constrained by training opponents. If opponents have bugs like a group of armies stuck somewhere, even though we can get a high reward in the

Table 3: The detailed evaluation result of LastOrder against AIIDE 2017 StarCraft AI competition bots without IO sync.

Bot name	Race	Games	Win rate %
Iron	Terran	303	98.68
PurpleWave	Protoss	303	97.35
LastOrder	Zerg	8484	83.06
Microwave	Zerg	303	49.67
Ximp	Protoss	303	31.91
LetaBot	Terran	303	28.48
IceBot	Terran	303	22.77
Arrakhammer	Zerg	303	15.18
Skynet	Protoss	303	15.13
Juno	Protoss	303	15.13
Steamhammer	Zerg	303	13.82
cpac	Zerg	303	12.5
AILien	Zerg	303	12.21
UAlbertaBot	Random	303	10.6
McRave	Protoss	303	9.54
Myscbot	Protoss	303	9.24
CherryPi	Zerg	303	7.59
Overkill	Zerg	303	6.27
Aiur	Protoss	303	3.29
Xelnaga	Protoss	303	2.96
KillAll	Zerg	303	1.97
GarmBot	Zerg	303	1.65
Tyr	Protoss	303	1.65
MegaBot	Protoss	303	1.64
ZZZKBot	Zerg	303	1.64
Sling	Zerg	303	1
TerranUAB	Terran	303	0.99
Ziabot	Zerg	303	0.99
ForceBot	Zerg	303	0.66

end, it is not a valuable reward and may lead the policy to a wrong direction. A better solution may be the self-play training. But the self-play training needs micro and macro codes of three races which may also be a big overhead.

Unit level control. Micro management in StarCraft is a multi-agent system. It is difficult for rule-controlled units to behave properly in different situations and cooperate with each other. But how to train this multi-agent model and react in real time at a relatively large scale is still an open question.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, 265–283.

AIIDE. 2018. Official AIIDE 2018 StarCraft Competition Results. http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2018/.

AIUR. 2018. AIUR for StarCraft AI Competitions. https://aiur-group.github.io/AIUR/.

Churchill, D. 2018a. Tournament Manager Software for StarCraft AI Competitions. https://github.com/davechurchill/StarcraftAITournamentManager.

Churchill, D. 2018b. UAlbertaBot for StarCraft AI Competitions. https://github.com/davechurchill/ualbertabot.

Hausknecht, M., and Stone, P. 2015. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527* 7(1).

Heinermann, A. 2018. BWAPI: Brood war api, an api for interacting with starcraft: Broodwar (1.16.1). https://github.com/bwapi/bwapi.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Horgan, D.; Quan, J.; Budden, D.; Barth-Maron, G.; Hessel, M.; Van Hasselt, H.; and Silver, D. 2018. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*.

Hostetler, J.; Dereszynski, E. W.; Dietterich, T. G.; and Fern, A. 2012. Inferring strategies from limited reconnaissance in real-time strategy games. *arXiv preprint arXiv:1210.4880*.

Hsieh, J.-L., and Sun, C.-T. 2008. Building a player strategy model by analyzing replays of real-time strategy games. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, 3106–3111. IEEE.

Kim, J.; Yoon, K. H.; Yoon, T.; and Lee, J.-H. 2010. Cooperative learning by replay files in real-time strategy game. In *International Conference on Cooperative Design, Visualization and Engineering*, 47–51. Springer.

Kong, X.; Xin, B.; Liu, F.; and Wang, Y. Effective master-slave communication on a multi-agent deep reinforcement learning system.

Kong, X.; Xin, B.; Liu, F.; and Wang, Y. 2017. Revisiting the master-slave architecture in multi-agent deep reinforcement learning. *arXiv preprint arXiv:1712.07305*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, 278–287.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games* 5(4):293–311.

OpenAI. 2018. OpenAI Five. https://blog.openai.com/openai-five/.

Robertson, G., and Watson, I. D. 2014. An improved dataset and extraction process for starcraft ai. In *FLAIRS Conference*.

Shao, K.; Zhu, Y.; and Zhao, D. 2018. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Sun, P.; Sun, X.; Han, L.; Xiong, J.; Wang, Q.; Li, B.; Zheng, Y.; Liu, J.; Liu, Y.; Liu, H.; et al. 2018. Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game. *arXiv preprint arXiv:1809.07193*.

Synnaeve, G.; Bessiere, P.; et al. 2012. A dataset for starcraft ai & an example of armies clustering. In *AIIDE Workshop on AI in Adversarial Real-time games*, volume 2012.

Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 140–147. IEEE.

Wu, Y., and Tian, Y. 2016. Training agent for first-person shooter game with actor-critic curriculum learning.