Proceedings of the Fourteenth Artificial Intelligence and
Interactive Digital Entertainment Conference (AIIDE 2018)

Modular Architecture for StarCraft II with Deep Reinforcement Learning

**Dennis Lee,* Haoran Tang,* Jeffrey O Zhang,
Huazhe Xu, Trevor Darrell, Pieter Abbeel**
University of California, Berkeley
{dennisl88, hrtang.alex, jozhang, huazhe_xu}@berkeley.edu, {trevor, pabbeel}@eecs.berkeley.edu
(* Equal contributions)

Abstract

We present a novel modular architecture for StarCraft II AI. The architecture splits responsibilities between multiple modules that each control one aspect of the game, such as build-order selection or tactics. A centralized scheduler reviews macros suggested by all modules and decides their order of execution. An updater keeps track of environment changes and instantiates macros into series of executable actions. Modules in this framework can be optimized independently or jointly via human design, planning, or reinforcement learning. We present the first result of applying deep reinforcement learning techniques to training two out of six modules of a modular agent with self-play, achieving 92% or 86% win rates against the "Harder" (level 5) built-in Blizzard bot in Zerg vs. Zerg matches, with or without fog-of-war.

Introduction

Deep reinforcement learning (deep RL) has become a promising tool for acquiring competitive game-playing agents, achieving success on Atari (Mnih et al. 2015), Go (Silver et al. 2016), Minecraft (Tessler et al. 2017), Dota 2 (OpenAI 2018), and many other games. It is capable of processing complex sensory inputs, leveraging massive training data, and bootstrapping performance without human knowledge via self-play (Silver et al. 2017). However, StarCraft II, a well recognized new milestone for AI research, continues to present a grand challenge to deep RL due to its complex visual input, large action space, imperfect information, and long horizon. In fact, the direct end-to-end learning approach has not even won the easiest built-in AI (Vinyals et al. 2017).

StarCraft II is a real-time strategy game that involves collecting resources, building production facilities, researching technologies, and managing armies to defeat the opponent. Its predecessor StarCraft has attracted numerous research efforts, including hierarchical planning (Weber, Mateas, and Jhala 2010) and tree search (Uriarte and Ontañón 2016) (see survey by Ontañón et al. (2013)). Most prior approaches focus on substantial manual designs, yet still unable to defeat professional players, potentially due to their inability to utilize game play experiences (Kim and Lee 2017).

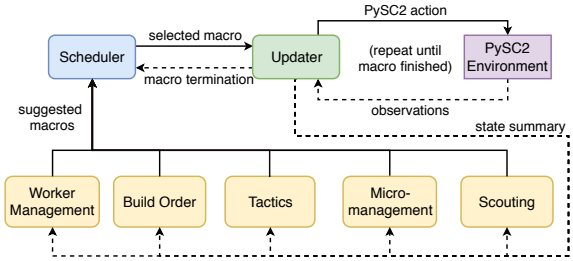

Figure 1: The proposed modular architecture for StarCraft II

We believe that deep RL with properly integrated human knowledge can effectively reduce the complexity of reinforcement learning, without compromising policy expressiveness or performance. To achieve this goal, we propose a flexible modular architecture that shares the decision responsibilities among multiple independent modules, including worker management, build order, tactics, micromanagement, and scouting (Figure 1). Each module can be manually scripted or handled by a neural network policy, depending on whether the task is routine and hence easy to handcraft, or highly complex and requires learning from data. All modules suggest macros (predefined action sequences) to the scheduler, which decides their order of execution. In addition, an updater keeps track of environment information and adaptively executes macros selected by the scheduler.

We further evaluate the modular architecture by reinforcement learning with self-play, focusing on important aspects of the game that can benefit more from massive training experiences, including build order and tactics. The agent is trained on the PySC2 environment (Vinyals et al. 2017) that exposes a challenging human-like control interface. We adopt an iterative training approach that first trains one module while others follow very simple scripted behaviors, and then replace the scripted component of another module with a neural network policy, which continues to train while the already trained modules remain fixed. We evaluate our agent playing Zerg v.s. Zerg against built-in bots on ladder maps, obtaining win rates 92% or 86% against the "Harder" bot, with or without fog-of-war. Furthermore, our agent generalizes well to held-out test maps and achieves similar performance.

Module	Responsibility	Current Design
Worker management	Ensure that resources are gathered at maximum efficiency	Scripted
Build order	Choose what unit/building/upgrade to produce	FC policy
Tactics	Choose where to send the army (attack or retreat)	FCN policy
Micromanagement	Micro-manage units to destroy more opposing units	Scripted
Scouting	Send scouts and track opponent information	Scripted + LSTM prediction

Table 1: The responsibility of each module and its design in our current version. FC = fully connected network. FCN = fully convolutional network.

Our main contribution is to demonstrate that deep RL and self-play combined with the modular architecture and proper human knowledge can achieve competitive performance on StarCraft II. Though this paper focuses on StarCraft II, it is possible to generalize the presented techniques to other complex problems that are beyond the reach of the current end-to-end RL training paradigm.

Related Work

Classical approaches towards playing the full game of StarCraft are usually based on planning or search. Notable examples include case-based reasoning (Aha, Molineaux, and Ponsen 2005), goal-driven autonomy (Weber, Mateas, and Jhala 2010), and Monte-Carlo tree search (Uriarte and Ontañón 2016). Most other research efforts focus on a specific aspect of the decision hierarchy, namely strategy (macromanagement), tactics, and reactive control (micromanaging). See the survey of Ontañón et al. (2013) and Robertson and Watson (2014) for complete summaries. Our modular architecture is inspired by hierarchical and modular designs that won past AIIDE competitions, especially UAlbertaBot (Churchill 2017), but features the integration of deep RL training and self-play instead of intense hardcoding.

Reinforcement learning studies how to act optimally in a Markov Decision Process to maximize the discounted sum of rewards $R = \sum_{t=0}^{T} \gamma^t r_t$ ($\gamma \in (0, 1]$). The book by Sutton and Barto (1998) gives a good overview. Deep reinforcement learning uses neural networks to represent the policy and/or the value function, which can approximate arbitrary functions and process complex inputs (e.g. visual information).

Recently, Vinyals et al. (2017) have released PySC2, a python interface for StarCraft II AI, and evaluated state-of-the-art deep RL methods. Their end-to-end training approach, although shows potential for integrating deep RL to RTS games, cannot beat the easiest built-in AI. Other efforts of applying deep learning or deep RL to StarCraft (I/II) include controlling multiple units in micromanagement scenarios (Peng et al. 2017; Foerster et al. 2017; Usunier et al. 2017; Shao, Zhu, and Zhao 2018) and learning build orders from human replays (Justesen and Risi 2017). To our knowledge, no published deep RL approach has succeeded in playing the full game yet.

Optimizing different modules can also be cast as a cooperative multi-agent learning problem. Apart from aforementioned multi-agent learning works on micromanagement, other promising methods include optimistic and hysteretic Q learning (Lauer and Riedmiller 2000; Matignon, Laurent, and Le Fort-Piat 2007; Omidshafiei et al. 2017), and centralized critic with decentralized actors (Lowe et al. 2017). Here we use a simple iterative training approach that alternately optimizes a single module while keeping others fixed, though incorporating multi-agent learning methods is possible and can be future work.

Self-play is a powerful technique to bootstrap from an initially random agent, without access to external data or existing agents. The combination of deep learning, planning, and self-play led to the well-known Go-playing agents AlphaGo (Silver et al. 2016) and AlphaZero (Silver et al. 2017). More recently, Bansal et al. (2018) has extended self-play to asymmetric environments and learns complex behavior of simulated robots.

Modular Architecture

Table 1 summarizes the role and design of each module. In the following sections, we will describe them in details for our implemented agent playing the Zerg race . Note that the design presented here is only an instance of all possible ways to implement this modular architecture. One can incorporate other methods, such as planning, into one of the modules as long as it works coherently with other modules.

Updater

The updater serves as a memory unit, a communication hub for modules, and a portal to the PySC2 environment.

To allow a fair comparison between AI and humans, Vinyals et al. (2017) define observation inputs from PySC2 as similar to those exposed to human players, including imagery feature maps of the camera screen and the minimap (e.g. unit type, player identity), and a list of non-spatial features such as the total amount of minerals collected. Because past actions, past events, and out-of-camera information are crucial for decision making but not directly accessible from current observations, the agent has to develop an efficient memory. Though it is possible to learn such a memory from experiences, we think a properly hand-designed set of memories can serve a similar purpose, while also reducing the burden on reinforcement learning. Table 3 lists example memories the updater maintains. Some memories (e.g. build queue) can be inferred from previous actions taken. Some (e.g. friendly units) can be inferred from inspecting the list of all units. Others (e.g. enemy units) require further processing PySC2 observations and collaborating with the scouting module.

The "notifications" entry holds any information a module wants to notify other modules, thus allowing them to com-

Module	Macro name and inputs	Executed sequence of macros or PySC2 actions
(All)	*jump_to_base* (base)	move_camera (base.minimap_location)
	select_all_bases	select_control_group (bases_hotkey)
Worker	*rally_workers* (base)	(1) *jump_to_base* (base), (2) *select_all_bases*, (3) rally_workers_screen (base.minerals_screen_location)
	inject_larva	(1) select_control_group (Queens_hotkey), (2) for each base: (2.1) *jump_to_base* (base), (2.2) effect_inject_larva_screen (base.screen_location)
Build order	*hatch* (unit_type)	(choose by unit_type, e.g. Zergling → train_zergling_quick)
	hatch_multiple_units (unit_type, n)	(1) *select_all_bases*, (2) select_larva, (3) *hatch* (unit_type) n times
	build_new_base	(1) base = closest unoccupied base (informed by the updater) (2) *jump_to_base* (base), (3) select_any_worker, (4) build_hatchery_screen (base.screen_location)
Tactics	*attack_location* (minimap_location)	(1) select_army, (2) attack_minimap (minimap_location)
Micros	*burrow_lurkers*	(1) select_army, (2) select_unit (lurker), (3) burrowdown_lurker_quick
Scouting	*send_scout* (minimap_location)	(1) select_overlord, (2) move_minimap (minimap_location)

Table 2: Example macros available to each module. A macro (italic) consists of a sequence of macros or PySC2 actions (non-italic). Information such as base.minimap_location, base.screen_location and bases_hotkey is provided by the updater.

Name	Description
Time	Total time (seconds) passed
Friendly bases	Minimap locations and worker counts
Enemy bases	Minimap locations (scouted)
Neutral bases	Minimap locations
Friendly units	Friendly unit types and counts
Enemy units	Enemy unit types and counts (scouted)
Buildings	All constructed building types
Upgrades	All researched upgrades
Build queue	Units and buildings in production
Notifications	Any message from or to modules

Table 3: Examples memories maintained by the updater

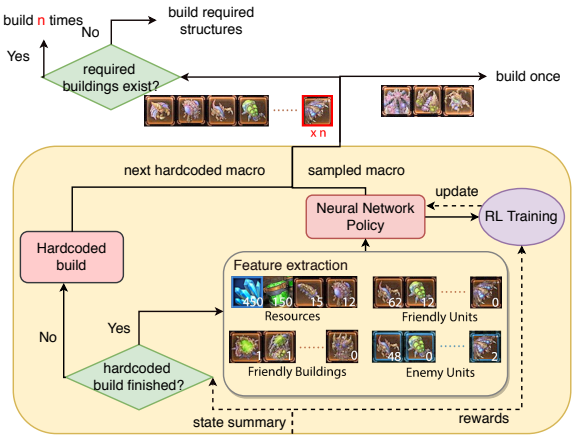

Figure 2: Details of our build order module.

municate and cooperate. For example, when the build order module decides to build a new base, it notifies the tactics module, which may move armies to protect the new base.

Finally, the updater handles communication between the agent and PySC2 by concretizing macros into sequences of PySC2 actions and executing them in the environment.

Macros

When playing StarCraft II, humans usually choose their actions from a list of subroutines, rather than from raw environment actions. For example, to build a new base, a player identifies an unoccupied neutral base, selects a worker, and then builds a base there. Here we name these subroutines as *macros* (examples shown in Table 2). Learning a policy to output macros directly can hide the trivial execution details of certain higher level commands, therefore allowing the policy to explore different strategies more effectively.

Build Order

A StarCraft II agent must balance our consumption of resources between many needs, including supply (population capacity), economy, combat units, upgrades, etc. The build order module plays the crucial role of choosing the correct thing to build. For example, in the early game, the agent needs to focus on building enough workers to gather resources, and while in the mid game, it should choose the correct types of armies that can beat the opposing ones. Though there exist numerous efficient build orders developed by professional players, executing one naively without adaptation can result in highly exploitable behavior. Instead of relying on complex if-else logic or planning to handle various scenarios, the agent's build order module can benefit effectively from massive game-play experiences. Therefore we choose to optimize this module by deep reinforcement learning.

Here we start with a classic hardcoded build order [1], as the builds are often the same towards the beginning of the game, and the optimal trajectory is simple but requires precisely timed commands. Once the hard-coded build is exhausted, a neural network policy takes control (See Figure 2). This policy operates once every 5 seconds. Its input consists of the agent's resources (minerals, gas, larva, and supply), its

[1] Exactly the first 2 minutes taken from https://lotv.spawningtool.com/build/56414/

building counts, its unit counts, and enemy unit counts (assuming no fog-of-war). We choose to exclude spatial inputs like screen and minimap features, because choosing what to build is a high-level strategic choice that depends more on the global information. The output is the type of unit or structure to produce. For units (Drones, Overlords, and combat units), it also chooses an amount $n \in \{1, 2, 4, 8, 16\}$ to build. For structures (Hatchery, Extractor) or Queen, it only produces one at a time. The policy uses a fully connected (FC) network with four hidden layers and 512 hidden units for each layer.

We also mask out invalid actions, such as producing more units than the current resources can afford, to enable efficient exploration. If a unit type requires a certain tech structure (e.g. Roaches need a Roach Warren) but it doesn't exist and is not under construction, then the policy will build the tech structure instead.

Tactics

Once our agent possesses an army provided by the build order module, it must learn to use it effectively. The tactics module handles map-level army commands, such as attacking or retreating to a specific location with a group of units. Though it is possible to hardcode certain tactics, we will show in the Evaluation section that a learned tactics can perform better.

In the current version, our tactics simply decides where on the minimap to move all of its armies towards. The input consists of 64×64 bitmaps of friendly units, enemy units (assuming no fog-of-war), and all selected friendly units on the minimap. The output is a probability distribution over minimap locations. The policy uses a three-layer Fully Convolutional Network (FCN) with 16, 32 and 1 filters, 5, 3 and 1 kernel sizes, and 2, 1 and 1 strides respectively. A softmax operation over the FCN output gives the probability over the minimap. The advantage of FCN is that its output is invariant to translations in the input, allowing the agent to generalize better to new scenarios or even new maps. The learned tactics policy operates every 10 seconds.

Scouting

Because the fog-of-war hides certain areas, enemy-dependent decisions can be very difficult to make, such as building the correct army types to counter the opponent's.

Our current agent assumes no fog-of-war during self-play training, but can be evaluated under fog-of-war at test time, with a scouting module that supplies missing information. In particular, the scouting module sends Overlords to several predefined locations on the map, regularly moves the camera to those places and updates enemy information. It maintains an exponential moving average estimate of enemy unit counts for the build order module, and uses a neural network to predict enemy unit locations on the minimap for the tactics module. The prediction neural network applies two convolutions with 16, 32 filters and 5, 3 kernel sizes to the current minimap, followed by an LSTM of 512 hidden units, whose output is reshaped to the same size of the minimap, followed by pixel-wise sigmoid to predict the probabilities of enemy units. Future work will involve adding further predictions beyond enemy locations and using RL to manage scouts.

Micromanagement

Micromanagement requires issuing precise commands to individual units, such as attacking specific opposing units, in order to maximize combat outcomes. Here we use simple scripted micros in our current version, leaving the learning of more complex behavior to future work, for example, by leveraging existing techniques presented in the Related Work. Our current micromanagement module operates when the updater detects that friendly units are close to enemies. It moves the camera to the combat location, groups up the army, attacks the location with most enemies nearby, and uses a few special abilities (e.g. burrowing lurkers, spawning infested terrans).

Worker Management

Worker management has been extensively studied for StarCraft: Brood War (Christensen et al. 2010). StarCraft II simplifies the process, so the suggested worker assignment (2 per mineral patch, 3 per vespene geyser) is often sufficient for professional players. We script this module by letting it constantly review worker counts of all bases and transferring excess workers to under-saturated mining locations, prioritizing gas over minerals. It also periodically commands Queens to inject larva at all bases, which is crucial for boosting the population.

Scheduler

The PySC2 environment places a restriction on the number of actions per minute (APM) to ensure a fair comparison between AI and human. Therefore when multiple modules propose too many macros at the same time, not all macros can be executed and a scheduler is required to order them by priority. Our current version uses little APM, so the scheduler simply cycles through all modules and executes the oldest macro proposals. When APM increases in the future, for example when complex micromanagement comes into play, a cleverer or even learned scheduler is required.

Training Procedure

Our agent is trained to play Zerg v.s. Zerg on the ladder map Abyssal Reef on the 4.0 version of StarCraft II. For most games, Fog-of-war is disabled. Note that built-in bots can also utilize full observations, so the comparison is fair. Each game lasts 60 minutes, after which a tie is declared.

Self-Play

We follow the self-play procedure suggested by Bansal et al. (2018) to save snapshots of the current agent into a training pool periodically (every 3×10^6 policy steps). Each game the agent plays against a random opponent sampled uniformly from the training pool. To increase the diversity of opponents, we initialize the training pool with a random agent and other scripted modular agents that use fixed build orders and simple scripted tactics. The fixed build

orders are optimized[2] to prioritize specific unique types and include *zerglings*, *banelings*, *roaches_and_ravagers*, *hydralisks*, *mutalisks*, *roaches_and_infestors*, and *corruptors_and_broodlords*. Zerglings are available to every build order. The scripted tactics attacks the enemy bases with all armies whenever its army supply is above 50 or its total supply is above 100. The agent never faces the built-in bots until test time.

Reinforcement Learning

If winning games is the only concern, in principle the agent should only receive a binary win-loss reward. However, we have found that the win-loss provides too sparse training signals and thus slows down training. Instead we use the supply difference (d_t) between the agent and the enemy as a reward function. A positive supply difference is often correlated with an advantageous status. Specifically, to ensure that the game is always zero-sum, the reward is the *change* in supply difference $r_t = d_t - d_{t-1}$ for each time step. Summing up all rewards yields a total reward equal to the end-game supply difference.

We use Asynchronous Advantage Actor-Critic (Mnih et al. 2016) to optimize the policies with 18 parallel CPU workers. The learning rate is 10^{-4} and the entropy bonus coefficient is 10^{-1} for build order, 10^{-4} for tactics (smaller due to a larger action space). Each worker commits a gradient update to the central parameter server every 40 policy steps (roughly 3 minutes in the game for build order and 6 minutes for tactics).

Iterative Training

One major benefit of the modular architecture is that modules act relatively independently and can therefore be optimized separately. We illustrate this by comparing iterative training, namely optimizing one module while keeping others fixed, against joint training, namely optimizing all modules together. We hypothesize that iterative training can be more effective because it stabilizes the experiences gathered by the training module and avoids the complex module-wise coordination during joint training.

In particular, we pretrain a build order module with a scripted tactics described in the Self Play section, and meanwhile pretrain a tactics module with a scripted build order (all Roaches). Once both pretrained modules stabilize, we combine them, freeze the tactics, and only train the build order. After build order stabilizes, we freeze its parameters and train tactics instead. The procedure is abbreviated "iterative, pretrained build order and tactics".

Evaluation

Videos our agent playing against itself and qualitative analysis of the tactics module are available on https://sites.google.com/view/modular-sc2-deeprl. In this section, we would like to analyze the quantitative and qualitative performance of our agent by answering the following questions.

[2]Most builds taken from https://lotv.spawningtool.com/build/zvz/

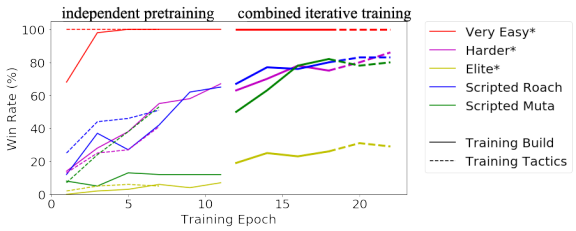

Figure 3: Win rates of our agent against opponents of different strengths. Asterisks indicate built-in bots that are not seen during training. 1 epoch = 3×10^5 policy steps.

Figure 4: Learned army compositions. Showing ratios of total productions of each unit type to the total number of produced combat units.

1. Does our agent trained with self-play outperform scripted modular agents and built-in bots?
2. Does iterative training outperform joint training?
3. How does the learned build order behave qualitatively? E.g. does it choose army types that beat the opponent's?
4. Can our agent generalize to other maps after being trained on only one map?

Quantitative Performance

Figure 3 shows the win rates of our agent throughout training, under the "iterative, pretrained build order and tactics" procedure. Pretrained build order and tactics can already achieve 67% and 41% win rates against the Harder bot. The win rate of combined modules increase to 86% after iterative training. Moreover, it outperforms simple scripted agents (also see Table 5), indicating the effectiveness of reinforcement learning.

Table 4 shows that iterative training outperforms joint training by a large margin. Even when joint training also starts with a pretrained bulid order, its stability quickly drops and results in 50% less win rate than iterative against the Harder bot. Pretraining both build order and tactics lead to better overall performance.

Qualitative Evaluation of the Learned Build Order

Figure 4 shows how our learned build order module reacts to different scripted opponents that it sees during training.

Training procedure	Hard	Harder	Very Hard	Elite
Iterative, no pretrain	79%	66%	13%	12%
Iterative, pretrained tactics	84%	73%	17%	8%
Iterative, pretrained build order	**86%**	81%	38%	24%
Iterative, pretrained build order and tactics	84%	**86%**	**58%**	**29%**
Joint, no pretrain	52%	25%	7%	7%
Joint, pretrained build order	60%	31%	9%	7%

Table 4: Comparison of final win rates (out of 100 matches) against built-in bots between different training procedures

	Average	Roach	Mutalisk	V. Easy	Easy	Medium	Hard	Harder	V. Hard	Elite
Modular (None)	68%	69%	44%	100%	92%	72%	79%	66%	13%	12%
Modular (Tactics)	62%	75%	31%	100%	100%	90%	84%	73%	17%	8%
Modular (Build)	77%	81%	42%	100%	96%	92%	**86%**	81%	38%	24%
Modular (Both)	**83%**	80%	**67%**	**100%**	**100%**	**99%**	84%	**86%**	**58%**	**29%**
Scripted Roaches	61%	–	18%	100%	100%	90%	71%	35%	5%	5%
Scripted Mutalisks	72%	**82%**	–	100%	98%	86%	76%	64%	15%	2%

Table 5: Comparison of win rates (out of 100 matches) against various opponents. Pretrained component in parenthesis. "V." means "Very".

Though our agent prefers the Zergling-Roach-Ravager composition in general, it can correctly react to the Zerglings with more Banelings, to Banelings with fewer Zerglings and more Roaches, and to Mutalisks with more hydralisks. Currently, the reactions are not perfectly tailored to the specific opponents, likely because the Zergling-Roach-Ravager composition is strong enough to defeat the opponents before they can produce enough units.

Generalization to Different Maps

Many parts of the agent, including the modular architecture, macros, choice of policy inputs, and neural network architectures (specifically FCN), are designed with certain prior knowledge that can help with generalization to different scenarios. We test the effect of prior knowledge by evaluating our agent against different opponents on maps not seen during training. These test maps have various sizes, terrains, and mining locations. The 4-player map Darkness Sanctuary even randomly spawns players at 2 out or 4 locations. Table 6 summarizes the results. Though our agent's win rates drop by 6% on average against Harder, it is still very competitive.

Opponent	AR	DS	AP
Scripted Roaches	81%	83%	78%
Hard	84%	85%	78%
Harder	86%	79%	81%
Very Hard	58%	43%	55%
Elite	29%	22%	30%

Table 6: Win rates (out of 100 matches) of our agent against different opponents on various maps. Our agent is only trained on AR. AR = Abyssal Reef. DS = Darkness Sanctuary. AP = Acid Plant.

Opponent	Hard	Harder	V. Hard	Elite
Win Rate	94%	92%	49%	60%

Table 7: Win rates (out of 100 matches) of our agent on Abyssal Reef with fog-of-war enabled

Testing under Fog-of-War

Though the agent was trained without fog-of-war, we can test its performance by filling missing information with estimates from the scouting module. Table 7 shows that the agent actually performs much better under fog-of-war, achieving 9.5% higher win rates on average, potentially because the learned build orders and tactics generalize better to noisy/imperfect information.

Conclusions and Future Work

In this paper, we give the first demonstration that proper combination of human knowledge and deep reinforcement learning can result in competitive StarCraft II agents. Certain techniques like modular architecture, macros, and iterative training can provide insights to dealing with other challenging problems at the scale of StarCraft II.

However, our current version is still far from beating the hardest built-in bots, let alone skilled humans. Many improvements are under research, including deeper neural networks, multi-army-group tactics, researching upgrades, and learned micromanagement policies. We believe that such improvements can eventually close the gap between our modular agent and professional human players.

Acknowledgements

This work was supported in part by the DARPA XAI program and Berkeley DeepDrive.

References

Aha, D. W.; Molineaux, M.; and Ponsen, M. 2005. Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning*, 5–20. Springer.

Bansal, T.; Pachocki, J.; Sidor, S.; Sutskever, I.; and Mordatch, I. 2018. Emergent complexity via multi-agent competition. *International Conference on Learning Representations*.

Christensen, D. B.; Hansen, H. O.; Juul-Jensen, L.; and Kastaniegaard, K. 2010. Efficient resource management in starcraft: Brood war. https://projekter.aau.dk/projekter/files/42685711/report.pdf.

Churchill, D. 2017. Ualbertabot. https://github.com/davechurchill/ualbertabot.

Foerster, J.; Farquhar, G.; Afouras, T.; Nardelli, N.; and Whiteson, S. 2017. Counterfactual multi-agent policy gradients. *arXiv preprint arXiv:1705.08926*.

Justesen, N., and Risi, S. 2017. Learning macromanagement in starcraft from replays using deep learning. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, 162–169. IEEE.

Kim, Y., and Lee, M. 2017. Intelligent machines humans are still better than ai at starcraft—for now. *MIT Technology Review*.

Lauer, M., and Riedmiller, M. 2000. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer.

Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, O. P.; and Mordatch, I. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, 6382–6393.

Matignon, L.; Laurent, G. J.; and Le Fort-Piat, N. 2007. Hysteretic q-learning: An algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, 64–69. IEEE.

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.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 1928–1937.

Omidshafiei, S.; Pazis, J.; Amato, C.; How, J. P.; and Vian, J. 2017. Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *International Conference on Machine Learning*, 2681–2690.

Ontañón, 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, 2018. https://blog.openai.com/openai-five/. Accessed: 2018-08-19.

Peng, P.; Yuan, Q.; Wen, Y.; Yang, Y.; Tang, Z.; Long, H.; and Wang, J. 2017. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*.

Robertson, G., and Watson, I. 2014. A review of real-time strategy game ai. *AI Magazine* 35(4):75–104.

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.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

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.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2017. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, volume 3, 6.

Uriarte, A., and Ontañón, S. 2016. Improving monte carlo tree search policies in starcraft via probabilistic models learned from replay data. In *AIIDE*.

Usunier, N.; Synnaeve, G.; Lin, Z.; and Chintala, S. 2017. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *International Conference on Learning Representations*.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Weber, B. G.; Mateas, M.; and Jhala, A. 2010. Applying goal-driven autonomy to starcraft. In *AIIDE*.