TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game

Peng Sun*[1], Xinghai Sun*[1], Lei Han*[1], Jiechao Xiong*[1], Qing Wang[1], Bo Li[1], Yang Zheng[1], Ji Liu[1,2], Yongsheng Liu[1], Han Liu[1,3], Tong Zhang[1]

[1]Tencent AI Lab [2]University of Rochester [3]Northwestern University

September 20, 2018

Abstract

Starcraft II (SCII) is widely considered as the most challenging Real Time Strategy (RTS) game as of now, due to large observation space, huge (continuous and infinite) action space, partial observation, multi-player simultaneous game model, long time horizon decision, etc. To push the frontier of AI's capability, Deepmind and Blizzard jointly present the StarCraft II Learning Environment (SC2LE) — a testbench for designing complex decision making systems. While SC2LE provides a few mini games such as *MoveToBeacon*, *CollectMineralShards*, and *DefeatRoaches* where some AI agents achieve the professional player's level, it is still far away from achieving the professional level in a *full* game. To initialize the research and investigation in the full game, we develop two AI agents — the AI agent TStarBot1 is based on deep reinforcement learning over flat action structure, and the AI agent TStarBot2 is based on rule controller over hierarchical action structure. Both TStarBot1 and TStarBot2 are able to defeat the builtin AI agents from level 1 to level 10 in a full game (1v1 *Zerg*-vs-*Zerg* game on the AbyssalReef map), noting that level 8, level 9, and level 10 are cheating agents with full vision on the whole map, with resource harvest boosting, and with both, respectively [1]. Specifically, TStarBot1 adopts a set of flat macro actions, over which

*Equal contribution.

[1]According to some informal discussions from the StarCraft II forum, level 10 builtin AI is estimated to be Platinum to Diamond [1], which are equivalent to top 50% - 30% human players in the ranking system of Battle.net Leagues [2].

a single controller is trained with Reinforcement Learning; TStarBot2 adopts macro-micro mixed actions organized hierarchically, and relies on controllers of hard-coded expert rules. To the best of our knowledge, this is the first public work to investigate the AI agent that is able to defeat the builtin AI in a StarCraft II full game. The code will be open sourced [3]. We hope the proposed framework can be beneficial for future research in several possible ways: 1) Be a baseline for hybrid system, where more and more learning modules will be gradually adopted and rules are still utilized to express logics that are hard to learn; 2) To generate trajectories for imitation learning; and 3) Be an opponent for self-play training.

1 Introduction

Recently, the marriage of Deep Learning [4] and Reinforcement Learning (RL) [5] gives rise to a breakthrough in decision making systems for non-trivial problems. Trained from scratch (or from a pre-trained model) and fed with (almost) raw observation features, Deep Reinforcement Learning (DRL) has shown impressive performance in a wide range of applications, including playing the board game GO [6, 7], playing video games (e.g., Atari [8], the first person shooting game Doom/ViZDoom [9, 10] or Quake/DeepmindLab [11], Dota 2 [12]), Robot Visuomotor Control [13], Robot Navigation [14, 15], etc. The learned policy/controller could work surprisingly well, and sometimes outperforms super-human [8, 7].

However, Starcraft II (SCII) [16], which is widely considered as the most challenging RTS game, still remains unsolved. In SCII, a human player has to manipulate tens to hundred of units[2] for multiple purposes, e.g., collecting two types of resource, expanding for additional resources, upgrading technologies, building other units, sending squads for attacking or defending, performing micro managements over each unit for a battle, etc. This is one important factor why SCII is more challenging than Dota 2, in which the total number of units needed to be controlled is up to 5 (manipulated by 5 players respectively). Figure 1 shows a screenshot of what the human player interacts with. The opponent is hidden to the player, unless the units from the opponent is in the view range of the player's units. So the player needs to send units to perform scouting on the opponent's strategy. All the decisions must be made in real time. In terms of designing AI agents, SC2 involves large observation space, huge action space, partial observation, multi-player simultaneous game, long time horizon decision, etc. All of these factors make SCII extremely challenging. To push the frontier of AI's capability, Deepmind and Blizzard jointly introduce the StarCraft II Learning Environment (SC2LE) [16]. Some recent results by Deepmind [16, 17] show that their AI

[2]In the 1 vs 1 game, the maximal number of moving units controlled by a player could exceed a hundred.

Figure 1: A screenshot of the SCII game when a human player is manipulating units.

agent can achieve the professional player's level in a few mini games, where the agent, for example, manipulates a *Marine* to reach a beacon (*MoveToBeacon*) or manipulates several *Marines* to defeat several *Roaches* (*DefeatRoaches*), etc. However, it is still far away from achieving the professional level in a *full* game.

To initialize the investigation in a full game, we restrict our study in the following setting: 1vs1 Zerg-vs-Zerg on the AbyssalReef map. We develop two AI agents — the AI agent TStarBot1 is based on deep reinforcement learning over flat actions and the AI agent TStarBot2 is based on rule controllers over hierarchical actions. Both TStarBot1 and TStarBot2 are able to defeat the builtin AI agents from level 1 to level 10 in a full game, noting that level 8, level 9, and level 10 are cheating agents with full vision on the whole map, with resource harvest boosting, and with both, respectively. Note also that according to some informal discussions from the StarCraft II forum, level 10 builtin AI is estimated to be Platinum to Diamond [1], which are equivalent to top 50% - 30% human players in the ranking system of Battle.net Leagues [2]. Specifically, TStarBot1 is based on a "flat" action modeling, which flattens the action structure and produces a number of discrete actions. In this way, it is immediately ready for any off-the-shelf RL algorithm that takes as input discrete actions. TStarBot2 is based on "deep" action modeling, which yields an action hierarchy manually specified. The "deep" modeling intuitively better captures the action dependencies and enjoys a multiplicative expression power. However, the training would be more challenging as complicated *hierarchical* RL may get involved. Noticing this

trading-off, in this preliminary work we adopt a rule based controller for TStarBot2.

To the best of our knowledge, this is the first public work to investigate the AI agents that are able to defeat the builtin AI in a Starcraft II full game. The code will be open sourced [3]. We hope the proposed framework can be beneficial for future research in several possible ways: 1) Be a baseline for hybrid system, where more and more learning modules will be gradually adopted and rules are still utilized to express logics that are hard to learn; 2) Generate trajectories for imitation learning; and 3) Be an opponent for self-play training.

2 Related Work

The RTS game StarCraft I has been taken as the platform for AI research for long, refer to [18, 19] for a review. However, most of the researches involve the searching algorithms or multi-agent algorithms, which cannot be directly applied to the full game. For example, many multi-agent reinforcement learning algorithms have been proposed to learn agents either independently [20] or jointly [21, 22, 23, 24] with communications to perform collaborative tasks, where a StarCraft unit (e.g., a *Marine*, a *Dragon Knight*, a *Zergling*, etc.) is treated as an agent. These methods are only able to handle the mini-game, which should be viewed as a snippet of the full game.

A vanilla A3C [25] based agent is tried in SC2LE for a full game [16], but is reported to perform poorly. Recently, relational neural network is proposed for playing the SCII game [17] using a player-level modeling, but the studies are carried on mini-games, other than a full game.

Historically, some rule based decision systems are successful in specific domains, e.g., MYCIN for medical diagnosis or DENTRAL for molecule discovery [26]. However, the only way they improve is by manually adding knowledge, lacking an ability to learn from data or by interacting with an environment.

Rule based bot is popular in video game industry. However, the focus there is to develop tools for code reuse (e.g., Finite State Machine or Behavior Tree [27]), not on how the rules can be combined with learning based methods. There exists recent work that tries to perform reinforcement learning or evolutional algorithm over Behavior Tree [28, 29], but the observation settings are tabular, and unrealistic for the large scale game like SCII.

Our macro action based agent (Section 3.2) is similar to the work of [30], where the authors adopt macro action for a customized mini RTS game. However, our macro action set is much larger and encodes more concrete rules for the execution, and is henceforth more realistic for SC2LE.

While implementing the hierarchical action based agent (Section 3.3), we are inspired by UAlbertaBot [19] for the modular design therein, which is also widely adopted in the literature of StarCraft I AI. In spirit, the hierarchical action is similar to the FeUdal network [31], but we do not pursue an end to end learning of the whole hierarchy. Moreover, we allow each action subset to have its own observation and policy, which should be a useful treatment that rules out noisy information, as is discussed in [32] when modeling the sub-task Q head therein.

3 The Proposed TStarBot Agents

Among the multiple aforementioned challenges, this work focuses on how to harness the huge action space, which, we argue, arises from its intrinsic complex structure. Specifically, it lies in the following aspects.

Hierarchy nature. The complex hierarchy nature seems always accompany with the long-horizon decision problems in RTS game. A human player usually summarizes his/her thinking in several levels: global strategies, local tactics, and micro executions. If a learning algorithm is unaware of the "thinking levels" (i.e., the action hierarchy) and works directly on the the massive number of basic atomic actions in the full game play, then it is inevitably a nightmare for both training and exploration during RL. For example, PySC2 [16] defines the action space over the low-level human user interface, involving hundreds of hot-keys and thousands of mouse-clicks over screen coordinates. Following this setting, even the state-of-the-art RL algorithm can only achieve success in playing toy mini-games that has much shorter horizon than the full game [17]. Although many researches have been devoted to automatically learning the Markovian Decision Process hierarchy [33, 34, 35, 36, 31, 37], none of them, unfortunately, can work efficiently on environments as complex as SCII. Therefore, how to utilize the hierarchy nature to shape a tractable decision space and narrow down the exploration, without including too much additional structure learning complexity, is a challenging task.

Hard-rules in SCII are difficult to learn. Another challenge of designing learning-based agent is the large number of "hard rules" in RTS game. These hard rules are "physics laws" and can in no means be violated. They are easily interpreted by human players through in-game textual instructions, but are difficult for learning algorithms to discover by pure trial-and-error. Consider a human player starting to learn to play StarCraft-II, he/she is instructed by a textual tutorial to first select a *drone* unit to build a *RoachWarren* unit before selecting a *larva* unit to produce a *Roach* unit, and so on. In this way, he might know

the logics:

- *RoachWarren is a prerequisite of producing a Roach.*
- *RoachWarren is built by a Drone.*
- *Roach is produced from a Larva.*
- *Producing a Roach requires 75 minerals and 25 gas.*
- *.....*

In SCII there are thousands of such dependency rules, constituting a technology dependency tree, abbreviated as *TechTree* (See also Section 3.1). *TechTree* serves as the most important prior knowledge that a human player should learn from textual tutorials or materials on the game interface, other than exploration through trial-and-error. The TechTree unaware learning algorithm must spend a huge amount of time to learn the hard rules, which is disastrous especially when the feedback signal is sparse and delayed (i.e., the *win/loss* reward received at the end of each game). Thus, in RTS games, it is important to think about how to design a mechanism to encode these hard game rules directly into the agent's prior knowledge, instead of relying on pure learning.

Uneconomical learning for trivial decision factors. It is also worth noting that despite the tremendous decision space of SCII, not all the decisions matter. In other words, a considerable amount of decisions are redundant in that they will have negligible effects on the game's final outcome. For instance, when a human player wants to build a *RoachWarren* during game, there are at least three decision factors he/she has to consider:

- Decision Factor 1: When to build it? (Non-trivial)
- Decision Factor 2: Which *Drone* builds it? (Trivial)
- Decision Factor 3: Where to build it? (Trivial)

A proficient player would conclude that: 1) the first decision factor is a non-trivial one since when to build a *RoachWarren* will have a considerable impact on the entire game process; 2) the second decision is trivial because any random *Drone* can do the work with negligible difference of building efficiency; and 3) the third one can also be taken as trivial as long as the target position is not too far away from the self-base and the *geometry defense* is not considered. Learning algorithms unaware of the factors would consume too many learning resources, which may dominate learning the non-trivial decisions. For example, an accurate

placing decision of "where to build" requires a selection among thousands of 2-D coordinates. It is uneconomic to invest too many learning resources for such trivial factors.

To address these challenges, we propose to model the action structure by hand-tuned rules. By doing so, the available actions are reduced to a tractable number, which turns out to be easier for designing our decision making system. In this line of thought, we implement two agents. One adopts macro action and reinforcement learning based controller (Section 3.2), while the other adopts macro-micro hierarchically actions and rule based controller (Section 3.3). The action execution may rely on a per-unit-control of SCII game, which is implemented in our PySC2 extension (Section 3.1).

3.1 Our PySC2 Extension

SC2LE [16] is the platform jointly presented by DeepMind and Blizzard. The game core library provided by Blizzard exposes a raw interface and a feature map interface. The DeepMind PySC2 environment further wraps the core library in Python and fully exposes the feature map interface. The purpose is to closely mimics the human control (e.g., the mouse click somewhere, or pressing some keyboard button), which causes a huge number of actions due to the complexity of the intrinsic structure in SC2. It thus poses non-trivial difficulty for a decision making system. Moreover, such a "player-level" modeling is inconvenient for "unit-level" modeling, especially when considering multi-agent style methodology. In this work, we make additional efforts to expose the unit control functionality and encode the technology tree.

Expose unit control. In our PySC2 fork, we additionally expose the raw interface of the SCII core library, which enables a per unit observation/manipulation. At each game step, all the units visible to the player (depending on whether enabling fog-of-war) can be retrieved. Each unit is fully described by a property list, including, e.g., the position, the health, etc. Such a raw unit array is made as observation returned to the agent. Meanwhile, a per unit action is allowed to control each unit. The agent is thus able to send raw action commands to interested individual unit (e.g., some unit moves to somewhere, some unit attacks some other unit, etc.). The definition of the unit and the per-unit-action can be found in the protobuf from SCII core library.

Encode the technology tree. In Starcraft II, the player needs particular units/buildings/techs as prerequisites for other advanced units/buildings/techs. Following UAlbertaBot [19], we formalize these dependencies into a technology tree, abbreviated as *TechTree* in our fork. We have collected the complete *TechTree* for *Zerg*, which gives the cost, building time, building ability, builder, prerequisites for each *Zerg* unit.

Besides the two additional functions described above, our PySC2 fork is fully compatible with the original Deepmind PySC2.

3.2 TStarBot1: A Macro Action Based Reinforcement Learning Agent

We illustrate in Figure 2 how the agent works. At the top, there is a single global controller, which will be learned by RL algorithm and make decisions over macro actions exposed to it. At the bottom, there is a pool of the macro actions, which hard-code prior knowledge of game rules (e.g. *TechTree*) and how the action is taken (e.g., which drone builds and where to build for a building action). It thus hides to the controller the trivial decision factors and executing details.

With this architecture, we relieve the learning algorithms from a heavy burden of directly handling a massive number of atomic operations, while still preserving most of the key decision flexibility of the full-game's macro strategies. Also, such an agent can be equipped with some basic knowledge of hard game rules even before learning. With such a mid-level abstracted and prior-knowledge enriched action space, the agent can learn fast from scratch and beat the most difficult built-in bots within $1 \sim 2$ days of training over a single GPU. More details are provided in the following subsections.

3.2.1 Macro Actions

We design 165 macro actions for the *Zerg*-vs-*Zerg* SCII full-game, as summarized in Table 1 (please refer to Appendix-I for the full list). As explained above, the purpose of the macro actions are on two-folds:

1. Encoding the game's intrinsic rules that are difficult to learn through only trial-and-error.
2. Hiding trivial decisions from the learning algorithm by random or scripted decision making.

Each macro action executes a meaningful elementary task, e.g., build a certain building, produce a certain unit, upgrade a certain technology, harvest a certain resource, attack a certain place, etc., and consists of a composition or a series of several atomic operations. With such an abstraction in action space, the high-level strategy towards winning an full game becomes easier to explore and learn. Some examples of macro actions are illustrated below.

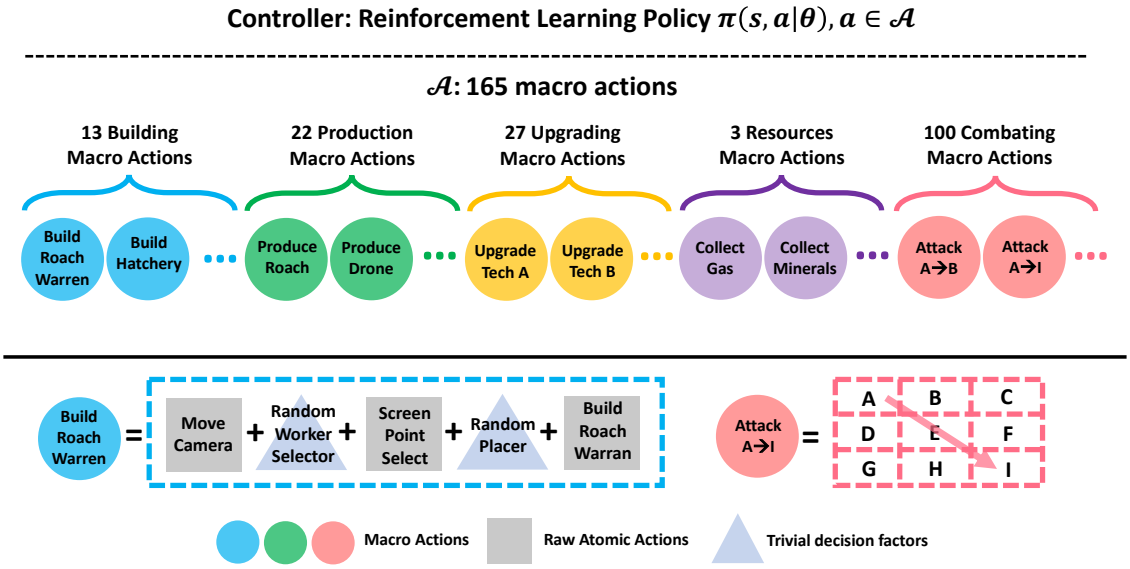

Figure 2: **Oerview of the agent based on macro action and reinforcement learning.** At the top: a learnable controller over the macro actions exposed from the bottom; At the bottom: a pool of 165 executable macro actions, which hard-code prior knowledge of game rules (e.g. TechTree) and hide to controller the trivial decision factors (e.g. building placement) and some execution details. The figure also illustrates the definition of two macro actions as examples: *BuildRoachWarren* and *ZoneAAttackZoneI*.

Table 1: **The summary of 165 macro actions**: their categories, examples and the hard-coded rules / knowledge. In the rightmost column, *TechTree* has been explained in 3.1; *RandUnit* refers to randomly selecting a subject unit; *RandPlacer* refers to randomly selecting a valid placement coordinate.

Action Category	#	Examples	Hard-coded rules / knowledge
Building	13	*BuildHatchery*,*BuildExtractor*	TechTree, RandUnit, RandPlace
Production	22	*ProduceDrone*, *MorphLair*	TechTree, RandUnit
Tech Upgrading	27	*UpgradeBurrow*, *UpgradeWeapon*	TechTree, RandUnit
Resources Harvesting	3	*CollectMinerals*, *InjectLarvas*	RandUnit
Combating	100	*ZoneBAttackZoneD*	Micro Attack/ Rally

Building Actions: Buildings are prerequisites for further unit production and tech upgrading in SCII. The building category contains 13 macro actions, each of which builds a certain *Zerg* building when taken. For example, the macro action *BuildSpawningPool*

builds a *SpawningPool* unit with a series of atomic *ui-actions* [3]: 1) *move_camera* to base, 2) *screen_point_select* a drone (the subject unit), 3) *build_spawningpool* somewhere in the screen. The serial atomic operations involve two internal decisions to make: 1) which *Drone* to build it? and 2) where to build it? Since these two decisions are usually considered to have little effect on the entire game process and outcome, we delegate them to some random-based and rule-based decision-makers, namely, a random *Drone* selector and a random spatial placer in this case. Noting that the random spatial placer has to encode the basic placement rules like: *Zerg* buildings can only be placed on the *Creep* zone; *Hatchery* has to be located near minerals for a fair harvesting efficiency. Besides, the *TechTree* rules such as "only *Drone* can build *SpawningPool*" are also encoded in this macro action.

Production & Tech Upgrading Actions: The unit production and tech upgrading largely shape the economy and technology development in the game. The production category contains 22 macro actions and the tech upgrading contains 27, each of which produces a certain type of units or upgrade a certain technology. The hard-coding of these macro actions goes similar to those of building actions described above, except that they do not need a spatial placer.

Resource Harvesting Actions: *Minerals*, *Gas*, and *Larvas* (*Zerg*-race only) are among the three key resources in SCII games. Their storage and collection speed can greatly affect the economy growth. We designed 3 corresponding macro actions: *CollectMinerals*, *CollectGas* and *InjectLarvas*. *CollectMinerals* and *CollectGas* assign a certain number of random workers (i.e. *Drone* in *Zerg*-race) to mineral shards or gas extractors, so that with these two macro actions, the workers can be re-allocated to different tasks, making the mineral and gas storage (or their ratio of storage) altered to meet certain needs. *InjectLarvas* simply orders all the idle queens to inject *Larvas*, to speed up the unit production procedure.

Combat Actions: How to design combat actions remains the most decisive part towards the outcome of one game, and it is also the most elaborate part to be abstracted into macro actions due to the potentially diverse macro combat strategies. For examples:

- Attack timing: e.g., rush, early harass, attack at the best timing windows.
- Attack routes: e.g., walk around narrow slopes which might constrain the attacking firepower.

[3] *ui-actions* refers to the actions of the *ui-control* interface in PySC2, resembling the human-player interface. In fact, we use in this project the *unit-control* interface (as described in Sec 3.1) which simplifies the execution path by allowing agents to directly push action commands to each individual unit without having to first highlight-select a subject unit before issuing a command to it.

- Rally positions: e.g., rally before attack in order for concentrated fire (note that various units might have different moving speed).

We attempt to represent such combat strategies by region-wise macro actions, which are defined as follows (demonstrated in Figure 2): we first divide the whole world map into 9 combat zones (named *Zone-A* to *Zone-I*), and an additional *Zone-J* for the whole world itself, resulting to 10 zones in total; based on the zone division, 100 ($= 10 \times 10$) macro actions are defined, with each macro action executing, e.g., "combat units in *Zone-X* start to attack *Zone-Y* if there are enemies there, otherwise, rally to *Zone-Y* and wait". For the micro attack tactics inside each macro action, in this work we simply hard-code the "hit-and-run" rule for each combat unit, i.e., the unit fires at the closest enemy and runs away upon low health. We leave it to future work the investigation of more sophisticated multi-agent learning for the micro tactics.

With the composition of these macro actions, a wide range of diverse macro combat strategies (although not of full flexibility) could be represented and fulfilled. For example, the selection of attacking routes can be fulfilled by a series of region-wise rally macro actions. With this definition, we avoid the use of a complex multi-agent setting.

Available Macro Action List. Not every pre-defined macro action described above is available at any time step. For example, *TechTree* constrains that some units/techs can only be built/produced/upgraded under certain conditions: e.g., enough storage of minerals/ gas/ food, existence of certain prerequisite unit/tech. The corresponding macro action should "do nothing" when these conditions are not satisfied. We maintain a list for such available macro actions at each time step, encoding the *TechTree* knowledge. This available action list masks those invalid actions at each step and henceforth eases the exploration. The list can also be taken as features.

3.2.2 Observations and Rewards

The observations come as a set of spatial 2-D feature maps and a set of non-spatial scalar features, extracted from the per-unit informations provided by the SCII game core and exposed in our PySC2 fork (see Section 3.1).

Spatial Feature Maps. Feature maps sized in $N \times N$ are rendered (not separated into mini-map and screen features as in the original PySC2 [16]), with each pixel indicating a certain statistic (e.g. the unit count of a certain type) of the world map region that corresponds to the feature map pixel. These feature maps include the count of commonly-used unit types for both self and enemies, as well as the count of units with certain attributes (e.g., "can-attack-ground", "can-attack-air").

Non-spatial Features. Scalar features include the amount of gas and minerals collected, the amount of food left, the counts of each unit types, etc. We also optionally include one-hot features of game progress and recently-taken actions when we don't use a recurrent model to keep track of the past information.

Rewards. We used the reward structure: ternary 1 (win) / 0 (tie) / -1 (loss) received at the end of a game, and the reward is always zero during the game. Although the reward signal is quite sparse and long-delayed, it works with our macro action space of tractable size.

3.2.3 Learning Algorithms and Neural Network Architectures

Based on the macro actions and observations defined above, the problem can be cast as a sequential decision process, where the discrete action space is in tractable size and the time horizon is shortened. At time step t, the agent receives an observation $s_t \in \mathcal{S}$ from the game environment, and chooses a macro action $a_t \in \mathcal{A}$ according to its policy, $\pi(a_t|s_t)$, a conditional probability distribution over $\mathcal{A}$, where $\mathcal{A}$ indicates the set of macro actions defined in Section 3.2.1; the selected macro action a_t is then translated into the game-core acceptable atomic actions by means of the aforementioned hand-tuned rules; after the atomic actions are taken, a reward [4] r_t and the next step observation s_{t+1} are received by the agent; this loop goes on and on until the end of a game.

Our goal is to learn an optimal policy $\pi^*(a_t|s_t)$ for the agent to maximize its expected cumulative rewards over all future steps. When we directly use the reward function defined in Section 3.2.2 without additional reward shaping, the optimization target is equivalent (when reward discount is ignored) to maximizing the agent's win-rate. We train our TStarBot1 agent to learn such a policy from scratch by playing against builtin AIs with off-the-shelf reinforcement learning algorithms (e.g. Dueling-DDQN [8, 38, 39] and PPO [40]), relying on a distributed rollout infrastructure. Details are described below.

Dueling Double Deep Q-learning (DDQN). Deep Q Network [8] first learns a parameterized estimation $\hat{Q}(s, a|\theta)$ of the optimal state-action value function (Q-function) $Q^*_\theta(s_t, a_t) = \max_\pi Q^\pi(s_t, a_t)$, where $Q^\pi(s_t, a_t) = \mathbb{E}_\pi[\sum_{i=t,\dots,T} \gamma^{i-t} r_i]$ is the expected cumulative future rewards under policy π. The optimal policy is then easily induced from the estimated optimal Q-function: $\pi(a_t|s_t) = 1.0$ if $a_t = \arg\max_{a\in\mathcal{A}} \hat{Q}(s_t, a|\theta)$, otherwise 0. Techniques such as replay memory [8], target network [8], double networks [38] and dueling architecture [39] are leveraged to reduce sample correlation, update target inconsistency, maximization bias and update target variance, thus increasing learning stability and sample efficiency. Due to the

[4] The reward is accumulated within the macro action's execution time, if the macro action lasts for multiple time-steps.

sparsity and long-delay of the rewards, we use a Mixture of Monte-Carlo (MMC) [41] return with the boostrapped Q-learning return as the Q update target, which further accelerates the reward propagation and stabilizes the training.

Proximal Policy Optimization (PPO). We also conducted experiments by directly learning a parametric form of stochastic policy $\pi(s_t, a_t|\theta)$ with Proximal Policy Optimization (PPO) [40]. PPO is a sample efficient policy gradient method, leveraging policy ratio trust region clipping to avoid the complex conjugate gradient optimization required to solve the KL-divergence constrained Conservative Policy Iteration problem in TRPO [42] . We used a truncated version of generalized advantage estimation [43] to trade-off the bias and variance of the advantage estimation. The available action list described in Section 3.2.1 is used to mask out unavailable actions and renormalizes the probability distributions over actions at each step.

Neural Network Architecture. We adopt multi-layer perception neural networks to parameterize the state-action value function, state value function and the policy function. While more complex network architectures could be considered (e.g., convolutional layers that extracts spatial features, or recurrent layers that compensates the partial observation), it is out the scope of this paper and we will focus on this simple network architecture.

Distributed Rollout Infrastructure. The SCII game core is CPU-intensive and slow for the rollout, resulting to a bottleneck during the RL training. To alleviate the issue, we build a distributed rollout infrastructure, where a cluster of CPU machines (called actors) are utilized to perform the rollout processes in parallel. The rollout experiences, cached in the replay memory of each actor, are randomly sampled and periodically sent to a GPU-based machine (called learner). We currently take 1920 parallel actors (with 3840 CPUs across 80 machines) to generate the replay transitions, at the speed of about 16,000 frames per second. This significantly reduces the training time (from weeks to days), and also improves the learning stability thanks to the increased diversity of the explored trajectories.

3.3 TStarBot2: A Hierarchical Macro-Micro Action Based Agent

The macro action based agent described in Section 3.2 might have limitations. Despite the macro actions can be grouped by functionality, the single controller has to work over the whole action set, where the actions are mutually exclusive at each decision step. Also, when predicting what action to take, the controller is fed into a common observation that is unaware of the action group. This amounts to unnecessary difficulties for training the controller, as undesired information may kick in for both observations and actions. On the other hand, the macro action alone does not expose control over the micro action (i.e., the

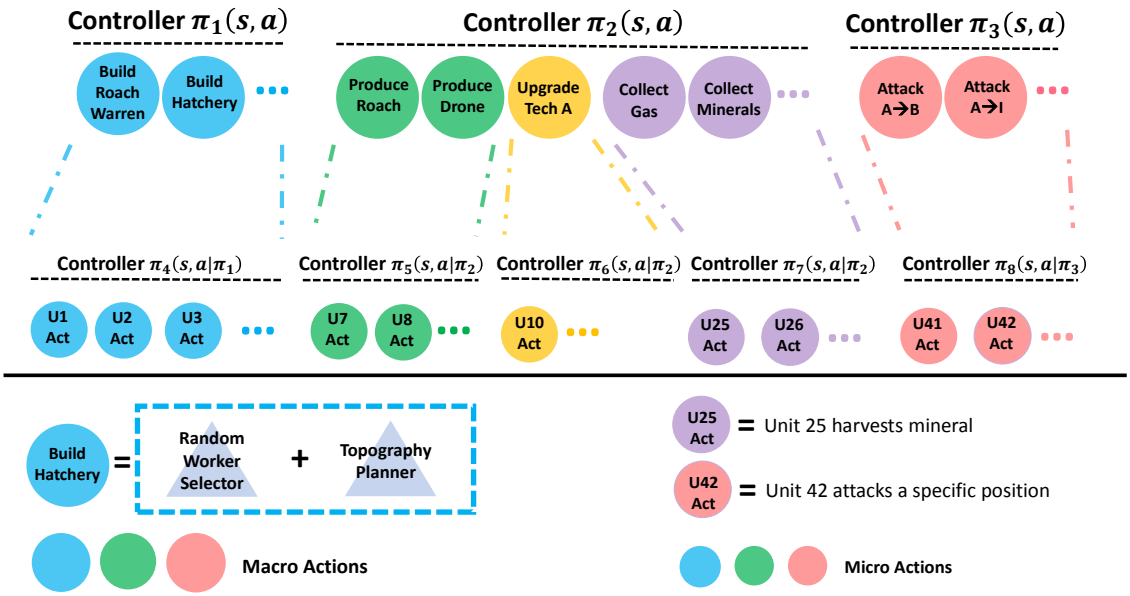

Figure 3: **Overview of the macro-micro hierarchical actions.** See the main text for explanations.

per-unit-control), which is inflexible when we want to adopt multi-agent style methodology.

We thus try a different set of actions, as in Figure 3. We adopt both macro actions and micro actions, organizing them in a two-tier structure. The upper tier corresponds to macro actions, which represent high-level strategies/tactics like "build RoachWarren near our main base" or "squad one attacks enemy base"; while the lower tier corresponds to micro actions, which correspond to low-level control over each unit like "unit 25 builds RoachWarren at a specific position" or "unit 42 attacks to a specific position". The whole action set is divided into subsets both horizontally and vertically. For each action subset, we assign it a separate controller that sees only the local action set, as well as the local observations that are relevant to the actions therein. At each time step, the controllers at the same tier can take simultaneous actions, while a downstream controller has to be conditioned on its upstream controller.

The advantage of such a hierarchical treatment is on two folds. 1) Each controller has its own observation/action space that the irrelevant information is ruled out, which is also adopted and discussed in [32] when modeling the sub-task Q head therein; 2) The hierarchy captures the action structure better, in particular the multiplicative expression power. This should be considered a more fine-grained modeling of the original action space.

Ideally, the controllers should be trained with RL either separately or jointly. However,

in current preliminary work we simply fill them with expert rules, intending to investigate whether it is potentially beneficial to introduce the hierarchical action set alone.

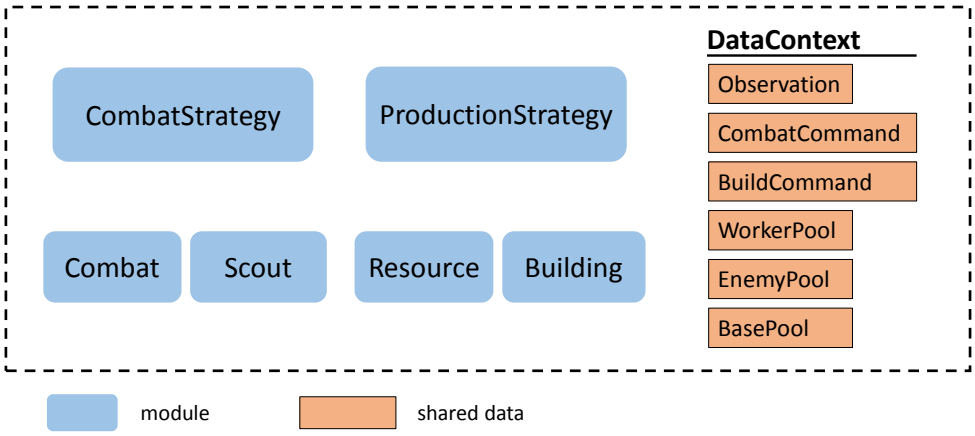

Figure 4: **Module diagram for the agent based on the Macro-Micro Hierarchical Action.** See the main text for explanations.

When writing the code, we encapsulate each controller as a module. The modules are organized in a similar way of UAlbertaBot, as shown in figure 4. The first tier modules (CombatStrategy, ProductionStrategy) only issue high-level commands (macro actions), while the second tier modules (Combat, Scout, Resource and Building) yield low-level commands (micro actions). All the modules are embedded into a DataContext, where each module can communicate with each other by sending/receiving messages and sharing customized data structures. Crucially, the game-play observation exposed by PySC2 is placed in the DataContext and henceforth visible to every module. This way, each module is able to extract from common observation the local observation that is relevant to its own action set. In the following we describe the modules in greater details.

3.3.1 Data Context

The DataContext module serves as a "black board" where each module exchanges informations. What are contained in the DataContext fall into the following categories:

1. Observation. The feature maps provided by PySC2, as well as the unit data structure of all active units at current game step, are exposed.

2. Pool. A pool is an array for a specific type of units, with associated properties/methods for the easy access of caller module. For instance, the WorkerPool is the array of all *Zerg Drones*. For another instance, the BasePool is the array of all *Zerg* bases, each item in the

pool being a BaseInstance. The BaseInstance is a customized data structure that records the Base (can be *Hatchery/Hive/Lair*), the associated *Drones*, *Minerals* and *Extractors* within a fixed range of Base, and a local coordinate system given by the geometrical layout of the minerals and the base.

3. Command Queue. High level commands are stored in queue and visible to all (lower-tier) modules. For instance, the commands issued by ProductionStrategy module are pushed in BuildCommand. Each command may look like "update a particular technology" or "harvest more minerals currently". Then the lower-tier module (respectively Building and Resource in this case) will pull from queue the command it recognizes and execute it by taking the rules and producing game core acceptable actions.

At each game step, the DataContext will update the Observations and the various Pools, while the Command Queues will be modified or accessed by other modules.

3.3.2 Combat Strategy

The combat strategy module makes high-level decisions to let the agent combat against the enemy in different ways. The module manipulates all the combat units[5] by organizing them into squads and armies. Each squad, which may contain one or multiple combat units, is expected to execute a specific task, such as harassing enemy base, cleaning the rock in the map, etc. Commonly, a small group of combat units with the same unit type is organized into a squad. An army contains multiple squads and it is given high-level strategic objective, e.g., attacking enemy, defending base, etc., and then specific commands are sent to each squad in the army. Each command, coupling with a squad-command pair, is then pushed into a combat strategy command queue (maintained in data context), which will be received and actually executed by the combat module.

Our implementation includes five high-level combat strategies:

- Rush: once a quad of small number of combat units has been built up, launch attack and keep sending squads to attack the enemy base.
- Economy First: keep collecting minerals and gas first. Launch attack after accumulating a large number of squads.
- Timing Attack: build up a strong army, consisting of *Roach* and *Hydralisk* squads, as quickly as possible and starts a strong attack.

[5]Currently, the combat units do not include *Drones* and *Overlords*.

- Reform: sort enemy bases and let the army attack the closest enemy base with priority. When approaching the target enemy base, stop the leading squads and let them wait other squads to stay together. Then, launch attack.

- Harass: set the combat strategy for the ground combat units as 'Reform'. Build up 2-3 squads of *Mutalisk* and assign target enemy base to each of them. Then, let the *Mutalisk* detour and harass the *Drones* of the target enemy base.

3.3.3 Combat

The combat module fetches command from the command queue and execute a specific action for each unit. It focuses on unit-level manipulation to effectively let each unit fight against the enemy. The combat module implements some basic human-like micro-management tactics, such as hit-and-run, cover-attack, etc., which can be deployed to all combat unit types. Specifically, an additional micro-management manger for each specific combat unit is implemented by taking fully use of their specific skills. For example, the roach micro-manager enables roaches to burrow down and run away from enemy to recover when they are weak; *Mutalisks* are coded to stealthily reach the enemy base and harass enemy's economic; *Lurkers* use carefully designed hit-and-run tactics combing with burrowing down and up; and queens can provide additional larvas and curve weak allies, etc. These micro-managements are organized into hierarchies and each part can be conveniently replaced with RL models.

3.3.4 Production Strategy

The production strategy module manages the building/unit production, tech upgrading and resource harvesting. The module controls the production of units and buildings by pushing production instructions to each base instance. The tech upgrading instructions and other specific instructions, such as *Zerg's Morph*, are pushed precisely to the target unit. Then the Building module will implement all the above production instructions. The resource harvesting command are highly abstract, the production strategy only need to determine what is prioritized, gas or mineral, according to the mineral / gas storage ratio. Then the Resource module will re-allocated the workers in each base instance after the priority instruction.

In the module, we maintain a building order queue as a short-term production planning. In most time, the manager will follow the order to produce items (units, buildings or techs) as long as the resource is enough and the prerequisites are satisfied. While in some special case (expanding a new base) or emergency situation, a more prioritized item will cut in front of the queue or even clear the whole queue and plan a new goal. When the queue is empty

(including at the beginning of game), a new short-term goal should be made by a strategy immediately.

When executing the production on at each game step, the prerequisites and resource requirement of current item will been checked according to the TechTree. The prerequisites of advanced items will be added into the queue automatically if not satisfied, and the current game step will be skipped if the resource requirement is not satisfied. Furthermore when the current item is ready to produce, the producing base instance is determined according to the different item type in this module.

By using different opening order and goal planning function, we have defined two different strategies for *Zerg* as following:

- RUSH: "Roach rush". It produces roaches at the beginning and upgrade tech *BURROW* and *TUNNELINGCLAWS* to give the *Roaches* the ability to burrow and move while burrowed and increases the health regeneration rate while burrowed. After that the strategy continuously produce Roaches and *Hydralisks*.

- DEF_AND_ADV: "Defend and Advanced Armies". This strategy produces many *SPINECRAWLERs* at the second base to defend and then gradually produce advanced armies. Almost all the types of combat units are included and the final ratio of each types is restricted by using a cap of unit number in the ultra goal dict.

3.3.5 Building

Building module receives and executes the high level commands issued by the Production Strategy, as described in Section 3.3.4. The "unary" commands (i.e., let some unit take act by itself), are straightforward to execute. Some "binary" commands deserve more explanations.

The command "Expand" will drag a drone from the specified base, send it to the specified "resource area" and start morphing a Hatchery, whose global coordinate had been pre-calculated by a heuristic method when accessing the map information for the first time.

The command "Building" will drag a drone, morph into the specified building at some position, whose coordinate is decided by a dedicated sub module, called Placer. In our implementation, we adopt a hybrid way for the building placement, i.e., some of the core buildings are placed in predefined positions, while others are placed randomly. Both these two types of positions are in BaseInstance coordinate system, and will be translated into global coordinate system when making game core acceptable action. Specifically, all the tech upgrading related buildings and the first 6 *SpineCrawlers* are pre-defined. Note that the layout of the 6 *SpineCrawlers* placement is critical (e.g., whether they are in diamond

formation or in rectangular formation), affecting the quality of the defense and whether we can survive the early rush of the opponent player. We tried several arrangements and found the diamond formation seems the best. The other buildings, including additional *SpineCrawlers*, will be placed randomly, where a uniformly random coordinate is repeatedly generated until it passes all validity checking (e.g., whether it is on *Zerg* creep, whether it overlaps with other buildings, etc.).

3.3.6 Resource

Resource module is in charge of harvesting minerals and gases by sending drones to either mineral shards or extractors. At each time step, this module must know whether the current working mode is "mineral first" or "gas first", which is a high level command, called "resource type priority", issued by the Production Strategy module. The goal of this module is to maximize the resource collecting speed, which can be a complex problem of control science. In our implementation, we adopt several rules to achieve this goal, which turns out be simpler yet effective. The underlying idea is to let every drone work and avoid any drone being idle. Specifically, we let the following rules be sequentially executed at each time step:

Intra-base rules. At each time step, the local drones associated with a BaseInstance will be rebalanced to harvest more minerals or more gases, depending on the "resource type priority" command. Note that for each base and extractor the SCII game core maintains two useful variables "ideal harvesters number", which means the suggested maximum number of drones working on it, and "assigned harvesters number", which means the number of drones working on it, respectively. By the two variables it is easy to decide whether there are under-filled/over-filled working drones for minerals/gases locally.

Inter-base rules. When a new branch base is about to finish, drag 3 drones from other base in advance to the new base. This improves the resource collecting efficiency by saving some waiting time. We found this trick is critical, especially when expanding the first branch base.

Global rules. It scans for each (possible) idle worker. If any, send it to the nearest base to harvest either mineral or gas, depending on the current working mode "resource type priority". Note that when minerals or extractors are exhausted and all local drones working on them become idle, the rules also ensure these idle workers be sent to nearby bases.

3.3.7 Scout

The Scout module intends to see as many enemy units as possible. With fog-of-war mode enabled, each own unit only has a confined view. As a result, many of the enemy units are

invisible, unless own units can approach and see them, i.e., the behavior of scouting.

In our implementation, we send *Zerg Drones* or *Overlords* to detect enemy units and store the seen units in EnemyPool, from which we can infer high level information, e.g., the location of the enemy main base or branch base, the current buildings the enemy has, etc. Such kind of information can be further used to infer enemy's strategy, and will be useful for the CombatStrategy or ProductionStrategy to make counter-strategy accordingly.

We define the following scout tasks.

Explore Task. Whenever there is new own *Overlord*, we send it to a mineral zone. This helps to overwatch the territory of the enemy and henceforth its economy. When attacked, the *Overlord* will retreat, otherwise it just stays at the target position.

Forced Task. We will send a Drone to enemy's first branch base. By doing so, we can find out, e.g., whether a lot of *Zerglings* have rallied that the enemy is about to perform a RUSH strategy at the early stage of the game play.

The activation of each task depends on the game play steps, and is configurable in the config file.

4 Experiment

Experimental results are reported for the two agents described in Section 3.2 and Section 3.3, respectively. We test the agent in a 1v1 *Zerg*-vs-*Zerg* full game. Specifically, the agent plays against builtin AI ranging from level 1 (the easiest) to level 10 (the hardest). The map we use is AbyssalReef[6], on which a vanilla A3C agent over the original PySC2 observations/actions was reported [16] to perform poorly when playing against builtin AI in a Terran-vs-Terran full game.

4.1 TStarBot1

The proposed macro-action-based agent TStarBot1(Section 3.2) is trained by playing against a mixture of builtin AIs in various difficulty levels: for each rollout episode, a difficulty level is sampled uniformly at random from level-1, 2, 4, 6, 9, 10 for the opponent builtin AI. We restrict TStarBot1 to take one macro action every 32 frames (i.e. about every 2 seconds), which shortens the time horizon to about $300 \sim 1200$ steps per game and reduces TStarBot1's APM (Actions Per Minute) to about $400 \sim 800$ that is more comparable with human players. In this preliminary experiments, we only use non-spatial features together

[6]This map is an official map widely used in world class matches.

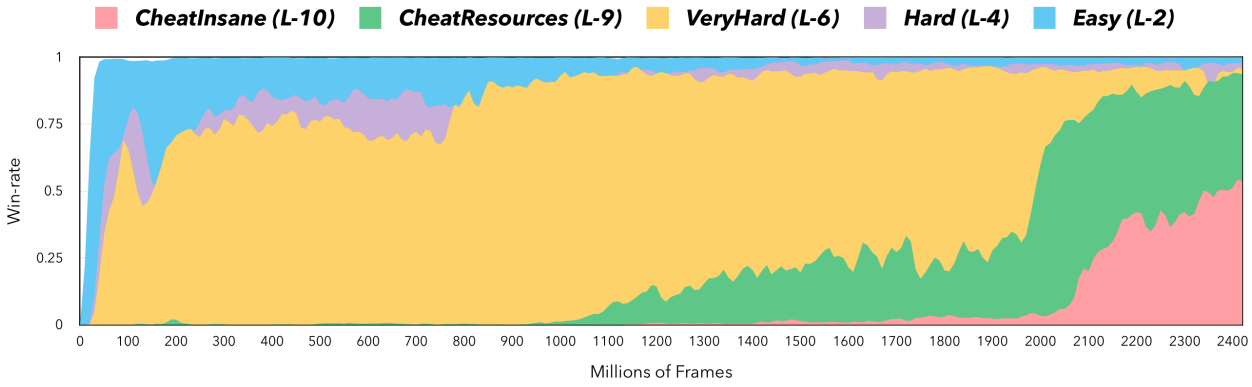

Figure 5: Learning curves of TStarBot1 with PPO algorithm. Note that TStarBot1 - PPO starts to defeat (at least 75% win-rate) *Easy* (Level-2) buildin AI at about 30M frames, *Hard* (Level-4) at about 250M frames, *VeryHard* (Level-6) at about 800M frames, *CheatResources* (Level-9) at about 2000M, and *CheatInsane* (Level-10) at about 3500M frames.

Table 2: **Win-rate (in %) of** TStarBot1 **and** TStarBot2 **agents, against builtin AIs of various difficulty levels**. For TStarBot1, results of DDQN, PPO, and a random policy are reported. Each win-rate is obtained by taking the mean of 200 games with different random seeds, with Fog-of-war enabled.

Difficulty Level IDs		L-1	L-2	L-3	L-4	L-5	L-6	L-7	L-8	L-9	L-10
Difficulty Level Descriptions		Very Easy	Easy	Medium	Hard	Harder	Very Hard	Elite	Cheat Vision	Cheat Resources	Cheat Insane
TStarBot1	RAND	13.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	DDQN	100.0	100.0	100.0	98.3	95.0	98.3	97.0	99.0	95.8	71.8
	PPO	100.0	100.0	100.0	100.0	99.0	99.0	90.0	99.0	97.0	81.0
TStarBot2		100.0	100.0	100.0	100.0	100.0	99.0	99.0	100.0	98.0	90.0

with simple MLP neural network. Also, in order to accelerate learning we prune the combat macro actions and only use *ZoneJ-Attack-ZoneJ*, *ZoneI-Attack-ZoneD*, *ZoneD-Attack-ZoneA*.

Table 2 reports the win-rates of TStarBot1 agent against builtin AI ranging from level 1 to level 10. Each reported win-rate is obtained by taking the mean of 200 games with different random seeds, where a tie is counted as 0.5 when calculating the win-rate. After about $1 \sim 2$ days of training with a single GPU and 3840 CPUs, the reinforcement learning agent (both DDQN and PPO) can win more than 90% of games against all built-in bots from level-1 to level-9, and more than 70% against level-10. The training and evaluation are both carried out with *Fog-of-war* enabled (no cheating).

Figure 5 shows the learning progress of TStarBot1 using the PPO algorithm. The

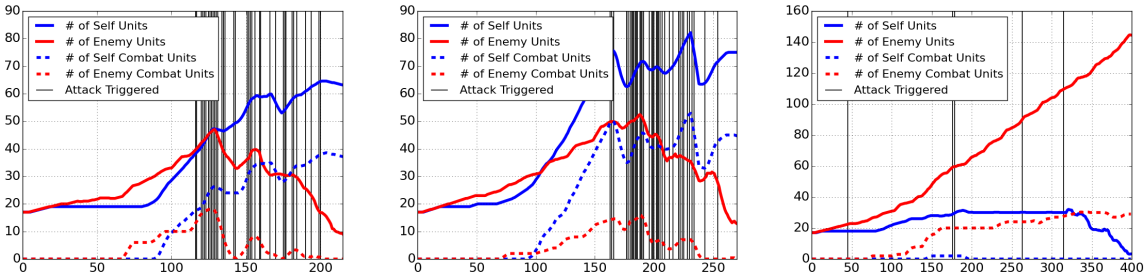

Figure 6: **The learned strategies about combat timing: *Rush* and *EconomyFirst*, for the** TStarBot1 **agent.** In each figure we plot several in-game statistics: self units count (blue solid curves), enemy units count (red solid curves), self combat-units count (blue dashed curves), enemy combat-unit count(red dashed curves), and combat timing (black vertical lines). The left and middle figures correspond to the learned RL policy, while the right figure corresponds to a random policy. The timing showed in the left figure resembles a human strategy called *Rush*, which launches attacks as soon as possible, even if there are only a small number of combat units available; The middle figure illustrates an *EconomyFirst* strategy, which launches the first attack only after having assembled a strong enough army.

curves show how the win-rate increases with the seen frames during training, each curve corresponding to a builtin AI in a certain difficulty level. Note that TStarBot1 learns and starts to defeat (at least 75% win-rate) *Easy* (level-2) builtin AI at about 30M frames (about 0.06M games), *Hard* (level-4) at about 250M frames (about 0.5M games), *VeryHard* (level-6) at about 800M frames (about 1.6M games), *CheatResources* (level-9) at about 2000M (about 4M games), and *CheatInsane* (level-10) at about 3500M frames (about 7M games).

After exploration and learning, the agent seems to acquire some intriguing strategies like human players. We demonstrate two strategies about the combat timing (i.e., when to trigger attacks) it learns, as in Figure 6: ***Rush***, which triggers attacks as soon as possible, even if there are only a small number of combat units available; ***EconomyFirst***, which keeps developing the economy and launches the first attack only after having assembled a strong army. Besides, we also observed that TStarBot1 tends to build $3 \sim 4$ bases to boost the economy growth and prefers sideways when planning for attacking routes.

4.2 TStarBot2

Table 2 shows the win-rate of the agent that adopts hierarchical macro-micro action and rule based controller (Section 3.3). Each reported win-rate is obtained by taking the mean

Table 3: **TStarBots vs. Human Players**. Each entry means how many games TStarBot1/TStarBot2 wins and loses. E.g., 1/2 means TStarBot "wins 1 game and loses 2 games".

#win/#loss	Platinum 1	Diamond 1	Diamond 2	Diamond 3
TStarBot1	1/2	1/2	0/3	0/2
TStarBot2	1/2	1/0	0/3	0/2

of 100 games with different random seeds, where a tie is counted as 0.5 when calculating the win-rate. For-of-war is enabled during the test. We can see that the agent is able to consistently defeat builtin AIs in all levels, showing the effectiveness of the hierarchical action modeling.

4.3 TStarBots vs. Human Players

In an informal internal test, we let TStarBot1 or TStarBot2 play against several human players ranging from Platinum to Diamond level in the ranking system of SCII Battle.net League. The setting remains the same as the above sub-sections, i.e., a Zerg-vs-Zerg full game on the map AbyssalReef with fog-of-war enabled. The results are reported in Table 3. We can see that both TStarBot1 and TStarBot2 are possible to defeat a Platinum (and even a Diamond) human player.

4.4 TStarBot1 vs. TStarBot2

In another informal test, we let the two TStarBots play against each other. We observe that TStarBot1 can always defeat TStarBot2. Inspecting the game-play, we find that TStarBot1 tends to use the *Zergling Rush* strategy, while TStarBot2 lacks anti-rush strategy and henceforth always loses.

It is worthy noting that although TStarBot1 can successfully learn and acquire strategies to defeat all the builtin AIs and TStarBot2, it lacks strategy diversity in order to consistently beat human players. In the aforementioned test with human players, TStarBot1 will be unable to win once the human player starts to know TStarBot1's preference for *Zergling Rush*. The insufficient strategy diversity might be caused by: 1) A lack of opponent diversity. Although the builtin AI is already equipped with several pre-defined strategies, their policy space is still too far away from the policy space formed by human players; 2) A lack of deep exploration. The production of advanced units are buried down very deeply in the tech tree, which is difficult and inefficient for a naive exploration (e.g., epsilon-greedy) to discover.

Self-play training and randomization techniques [12] seem to be promising to alleviate theses issues, and we leave it for future work.

5 Conclusions and Future Work

For SC2LE, we model the structural action space by hand-tuned rules, which reduces the number of actions to a tractable number. An agent based on flat action modeling and a reinforcement learning controller can acquire a reasonably high win-rate against builtin AI, while another agent adopting hierarchical action modeling and rule based controller can consistently win the builtin AI. In the future, we will work towards a unified approach: more carefully hand-tuned action hierarchy will be adopted, where each action set is assigned a separate controller with its own observation space and action space. All the controllers will be learned either separately or jointly. We are interested in whether such a treatment of the action space can boost a conventional RL algorithm to learn a good policy for the SCII 1v1 full game.

Acknowledgement

It is grateful that our colleagues Yan Wang and Lei Jiang, and a volunteer Yijun Huang participate the user study to test our AI agents.

References

[1] Starcraft ii forum. `https://eu.battle.net/forums/en/sc2/topic/853485381`. Accessed August 30, 2018.

[2] Liquipedia - ranking system. `https://liquipedia.net/starcraft2/Battle.net_Leagues`. Accessed August 30, 2018.

[3] Tstarbots. `https://the/link/will/be/available/in/the/next/version/of/this/manuscript/coming/soon`.

[4] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[5] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[6] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[9] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. *arXiv preprint arXiv:1605.02097*, 2016.

[10] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *International Conference on Learning Representations*, 2017.

[11] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[12] Open ai five. `https://blog.openai.com/openai-five/`. Accessed August 30, 2018.

[13] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[14] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. *International Conference on Robotics and Automation*, 2017.

[15] Fereshteh Sadeghi and Sergey Levine. (cad)2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[16] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[17] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

[18] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. 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, 2013.

[19] David Churchill. *Heuristic Search Techniques for Real-Time Strategy Games*. PhD thesis, PhD thesis, University of Alberta, 2016.

[20] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents.

[21] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. *arXiv preprint arXiv:1705.08926*, 2017.

[22] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*, 2016.

[23] Peng Peng, Ying Wen, Yaodong Yang, Quan Yuan, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017.

[24] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252, 2016.

[25] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[26] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[27] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

[28] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ogren. Towards a unified behavior trees framework for robot control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5420–5427. IEEE, 2014.

[29] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.

[30] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors,

Advances in Neural Information Processing Systems 30, pages 2659–2669. Curran Associates, Inc., 2017.

[31] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549, 2017.

[32] Harm Van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 5392–5402, 2017.

[33] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[34] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[35] Behzad Ghazanfari and Matthew E Taylor. Autonomous extracting a hierarchical structure of tasks in reinforcement learning and multi-task reinforcement learning. *arXiv preprint arXiv:1709.04579*, 2017.

[36] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*, pages 3486–3494, 2016.

[37] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

[38] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[39] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[41] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[42] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[43] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

Appendix I: List of Macro Actions

Table 4: List of all macro actions (for *Zerg* race only)

Categories	Macro Actions
Building	*BuildExtractor*
	BuildSpawningPool
	BuildRoachWarren
	BuildHydraliskDen
	BuildHatchery
	BuildEvolutionChamber
	BuildBanelingNest
	BuildInfestationPit
	BuildSpire
	BuildUltraliskCaven
	BuildNydusNetwork
	BuildSpineCrawler
Production	*ProduceDrone*
	ProduceZergling
	ProduceRoach
	ProduceHydralisk
	ProduceViper
	ProduceMutalisk
	ProduceCorruptor
	ProduceSwarmHost
	ProduceInfestor
	ProduceUltralisk
	ProduceOverlord
	ProduceQueen
	ProduceNydusWorm
	MorphLurkerDen
	MorphLair
	MorphHive
	MorphGreaterSpire
	MorphBaneling
	MorphRavager
	MorphLurker
	MorphBroodlord
	MorphOverseer

Categories	Macro Actions
Upgrading	*UpgradeBurrow*
	UpgradeCentrificalHooks
	UpgradeChitionsPlating
	UpgradeEvolveGroovedSpines
	UpgradeEvolveMuscularAugments
	UpgradeGliareConstituion
	UpgradeInfestorEvergy
	UpgradeNeuralParasite
	UpgradeOverlordSpeed
	UpgradeTunnelingClaws
	UpgradeFlyerArmorsLevel-1
	UpgradeFlyerArmorsLevel-2
	UpgradeFlyerArmorsLevel-3
	UpgradeFlyerWeaponLevel-1
	UpgradeFlyerWeaponLevel-2
	UpgradeFlyerWeaponLevel-3
	UpgradeGroundArmorsLevel-1
	UpgradeGroundArmorsLevel-2
	UpgradeGroundArmorsLevel-3
	UpgradeZerglingAttackSpeed
	UpgradeZerglingMoveSpeed
	UpgradeMeleeWeaponsLevel-1
	UpgradeMeleeWeaponsLevel-2
	UpgradeMeleeWeaponsLevel-3
	UpgradeMissileWeaponsLevel-1
	UpgradeMissileWeaponsLevel-2
	UpgradeMissileWeaponsLevel-3
Harvesting	*CollectMinerals*
	CollectGas
	InjectLarvas
Combating	*ZoneA-Attack-ZoneB*
	ZoneA-Attack-ZoneC
	.....
	ZoneJ-Attack-ZoneJ