Enhancing Sample Efficiency of Deep Reinforcement Learning to Master the Mini-games of StarCraft II

ZheJie Hu[1,a)] Tomoyuki Kaneko[2,b)]

Abstract: Recently, a variety of challenging domains have been mastered through deep reinforcement learning methods, and these methods also show great potential in real-world applications, like self-driving and automated manufacturing. However, most of deep reinforcement learning methods still suffer from requiring a large quantity of interaction samples. In this work, we apply Self-Imitation Learning to pick up past useful experiences that are similar to expert demonstrations to improve sample efficiency, and thus drive deeper explorations and obtain more rewards.

Keywords: Deep Reinforcement Learning, Self-Imitation Learning, GPU Asynchronous Advantage Actor-Critic, StarCraft II

1. Introduction

Recently, deep reinforcement learning methods have contributed to mastering a variety of challenging domains, such as Atari 2600 benchmarks [2] and the game of Go [9]. However, most deep reinforcement learning methods still suffer from requiring a large quantity of samples obtained from interaction, which indicates weak exploration and many oscillations during learning. Particularly, when training the agents to learn video games with large state and action space, such as StarCraft II, much trial-and-error is required for tuning policy to approximate the optimal policy. Intuitively, issues of low sample efficiency can be mitigated via better guidance of exploration, which can result in better exploitation. Some studies attempted to improving sample efficiency through using expert demonstrations [1], but the agents may perform poorly when encountering unseen states. Other studies have shown the advantages of experience replay [12] [8] in exploration, but extra cost for storing the experiences are required. In this work, we first apply GA3C to train agents to learn all mini-games of StarCraft II. Then we incorporate Self-Imitation Learning (SIL) [6] into GA3C to pick up past good experiences to improve sample efficiency. Our work has three main contributions as follows:

- Our proposed method, which is called GA3C+SIL, holds advantages of faster convergence and deeper exploitation than GA3C.
- This work updates our previous work [5] by augmenting entropy regularization in the policy loss.
- We design a new neural network structure that is inspired from study [11]. This well-designed neural network can deal with the information obtained in observation of StarCraft II better and help to improve the performance of agents.

[1] Graduate School of Interfaculty Initiative in Information Studies, the University of Tokyo, Japan
[2] Interfaculty Initiative in Information Studies, the University of Tokyo, Japan
[a)] ko-setsushou@g.ecc.u-tokyo.ac.jp
[b)] kaneko@acm.org

2. Background

2.1 StarCraft II Learning Environment

StarCraft II Learning Environment (SC2LE) was developed by both DeepMind and Blizzard Entertainment and is open sourced [10]. It consists of three parts: Linux StarCraft II Library, StarCraft II raw API and PySc2. Linux StarCraft II Library is a program of game itself, and is only available for Linux system. StarCraft II API is an interface that provides full external control of StarCraft II and critical for creating scripted bots. PySc2 is a Python environment that wraps the StarCraft II API to ease the interaction between Python RL agents and StarCraft II. In this work, we use PySc2 to serve as a bridge for interaction between the agents and StarCraft II environment.

2.1.1 State Space

The state (observation) provided by PySc2 consists of four main different parts, namely *screen*, *minimap*, *player information* and *available actions*. The first two parts are taken as spatial observations that consist of a group of feature layers. Such layers depict specific game feature as unit types, hit points and unit density that are shaped at N (weight) x M (height) pixels. Thanks

to the API provided by PySc2, N and M are configurable. *Screen* is a detailed view of local camera (or game screen), which is a subsection of the whole game map. *Minimap* is a coarse representation of the state of the whole game map. *Player information* contains most of non-spatial observations, which is denoted as an 1-D vector. Each element in this 1-D vector represents a non-spatial feature, such as the amount of resources gathered, the amount of supply that has been used and build-order queue. *Available actions* is a set of scalars, which corresponds to the indexes of actions that are currently available. There are totally 17 different feature layers in *screen* and 7 different feature layers in *minimap*. To our best knowledge, the scale of state space of StarCraft II is larger than most video games and all classic board games, thus large state space is a main challenge for the agents to master StarCraft II games.

2.1.2 Action Space

In StarCraft II, when the agents take actions, additional arguments are required. Because of this, we followed the study [10] and divided a whole action a into two parts, one is the action identifier a_0 and the other one is a sequence of its arguments $(a_1, a_2..a_L)$, where L depends on action identifier a_0. There are totally 549 different action identifiers with 13 possible types of arguments provided by PySc2 of version 2.0.2. In this work, we classify all the arguments into two categories: *spatial argument* and *non-spatial argument*. *Spatial arguments* ($[x_0, y_0]$ or $[(x_0, y_0), (x_1, y_1)]$) represent a set of all coordinates of *screen* or *minimap*. In StarCraft II, coordinate is required in many situations. For example, when we want a unit to move to a specific point of *screen*, the *move_to* command with corresponding coordinate of *screen* is needed. Additionally, not all of action identifiers are available at each time step. For example, if a unit is not selected, then the move command is not available to this unit. Therefore, *available actions* are included in the input for an agent when learning StarCraft II.

2.1.3 Mini-game

Study of SC2 have implemented several state-of-the-art deep RL algorithms on playing full game of StarCraft II, but no agents can even defeat the easiest built-in AI [10]. Therefore, in order to divide a big challenge into pieces, study of SC2 has offered seven mini-games, which address different parts of full game of StarCraft II. These mini-games have been well designed in the following aspects:

- Initial state is reasonable for the agents to do exploration during the rest of game.
- Preset reward structure can help to distinguish one policy from another policy, and thus the optimal policy is guaranteed to be found.

In this work, we focus on all the mini-games of StarCraft II to evaluate our proposed method–GA3C+SIL.

2.2 A3C and GA3C

A3C is an on-policy algorithm based on the advantage actor-critic algorithm. It deploys multiple actors running in parallel, each with its own copy of the environment. The critic is responsible for policy update. The diversity of actors experiences contribute to deeper exploration in different parts of the environment, thus reducing the correlation between samples to stabilize learning. However, in A3C, gradient updates can be computed on CPU only, which is slower than GPU in mathematical calculation, and can therefore be the bottleneck of training.

In order to accelerate gradient update, GA3C has shown us an effective way to extend A3C by utilizing GPU. It separates gradient update from interacting with the environment, and defines two kinds of threads that are called *Predictor* and *Trainer* to serve as the sample pipelines between *Agents* (CPU) and DNN (deep neural network, GPU). In GA3C, *Agents* only interact with their own environment like the actors do in A3C, but *Agents* do not keep their own copy of the deep neural network model. The *Predictor* receives observations dequeued from a prediction queue and returns policy generated from the DNN for next action. *Trainer* receives a batch of experiences from the *Agents* to update parameters of DNN. Unlike A3C that keeps a single and central DNN on CPU, GA3C places its unique central DNN on GPU and completes all gradient update on GPU. For both A3C and GA3C, the way to calculate the gradient of value function and policy gradient function are the same.

2.3 Prioritized Experience Replay

Prioritized Experience Replay (PER) [8] is an effective way to store past experience with a priority into a cache called replay buffer. The priority of each experience is related to the TD-error of RL methods [8]. For example, when combining Q-learning with PER, the priority of each experience is defined as the absolute value of TD error. Then, the agent sample experiences with probabilities proportional to their attached priority from replay buffer. PER can be incorporated into many RL methods, such as Q-learning and A3C. Although it requires a large amount of memory to store past experiences and extra cost of calculating priorities, PER actually improves performance of many actor-critic algorithms [12] [3]. In this work, we desire to apply PER to enhance exploitation of past experiences to improve

sample efficiency further.

2.4 Self-Imitation Learning

If the agents learn good past transitions frequently, the sample efficiency is expected to be improved. SIL algorithm makes use of advantage function of A2C [7] to determine whether a past transition is worth being learned or not. For example, given a past transition, a positive advantage value suggests that this transition can help to gain more rewards and is worth being learned twice or more times. When combined with PER, the priority of each transition is equal to its corresponding advantage that is clipped as $A_{sil} = \min(\max(\varepsilon, A), 1)$ (ε is a very small positive constant). Since sampling probability of each transition is proportional to its attached priority, better past transitions will be learned more frequently, and thus sample efficiency is expected to be improved further. Otherwise, as sampling of SIL only requires the advantage value, we can theoretically incorporate SIL into many advantage actor-critic based algorithms, such as A3C and GA3C.

3. Proposed Methods

This section describes our proposed methods. We first apply GA3C as the basic algorithm to training agents in SC2LE, and then we describe our implementation of GA3C+SIL in the latter part of this section.

Due to the complexity of StarCraft II, training time can cost a lot in some mini-games. Therefore, we design a simple neural network architecture. In order to compare sample efficiencies of different deep-RL methods implemented in this work, we also limit the amount of episodes for each mini-game.

3.1 Proposed GA3C

In the original GA3C, the agents are trained to learn Atari games. However, in our work, GA3C is applied to train agents to learn mini-games of StarCraft II. Thus, we modified the part of environment in GA3C. Figure 1 sketch the architecture of proposed GA3C.

In SC2LE, Agents take actions according to the policy that is brought from Deep Neural Network (DNN), and state and reward will be generated. For prediction, generated states are enqueued into a *Predictor queue* and will be dequeued when a *Predictor* does prediction. For Training, composition of generated states, rewards and corresponding actions can be taken as one experience. Such experiences are enqueued into a *Trainer queue* and will be dequeued when a *Trainer* does training. All the gradient update is executed by GPU in DNN. In a conclusion, the proposed GA3C is almost the same to the original GA3C apart from training environment.

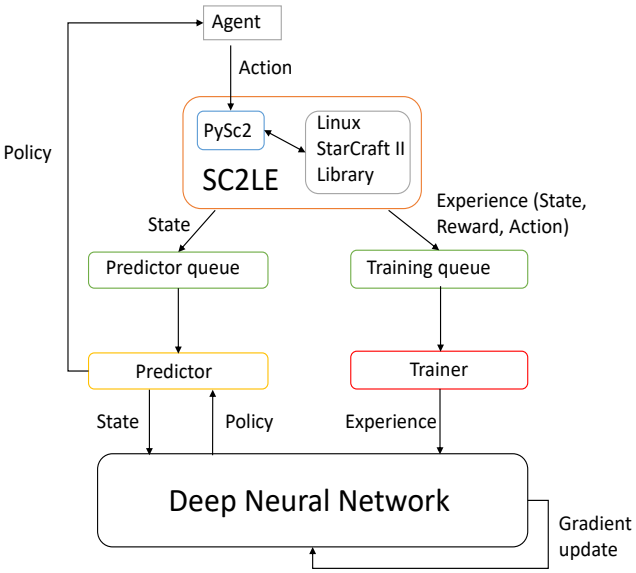

Fig. 1 Architecture of the proposed GA3C applied to SC2LE

3.2 GA3C with Self-Imitation Learning

In the original study of SIL, SIL is combined with A2C algorithms. However, as described in Section 2.3, SIL determines the priorities of past experiences according to the corresponding advantage values, and thus gives us a way to incorporate SIL into any advantage actor-critic based algorithm. In this work, we incorporate SIL into GA3C, which is denoted as GA3C+SIL. Figure 2 shows the architecture of proposed GA3C+SIL. This architecture is similar to the one of our previous work [5], but three things are modified to achieve better performance.

- In our previous work, we define a new queue called *RecorderQueue* that is not defined in study of original GA3C. This queue is used to receive experiences from agents and send past experiences to replay buffer. In this work, we implement this queue on receiving experiences from agents only, and we also delete thread of *Trainer* and define a new thread called *Recorder* to send experiences to both the deep neural network model and the replay buffer. *Recorder* can also receive sampled experiences from the replay buffer.
- In our previous work, we have only tested GA3C+SIL and GA3C on one mini-game called *DefeatRoaches* only. In this work, we conduct more experiments on more different mini-games of StarCraft II to compare performances of agents trained GA3C+SIL and GA3C respectively.
- In our precious work, we made mistakes of missing entropy regularization when incorporating SIL into GA3C, so our results are not as well as ex-

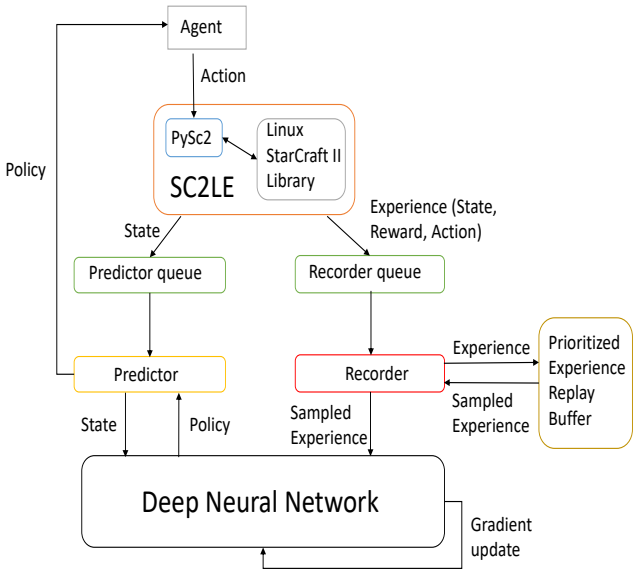

Fig. 2 Architecture of the proposed GA3C+SIL applied to SC2LE

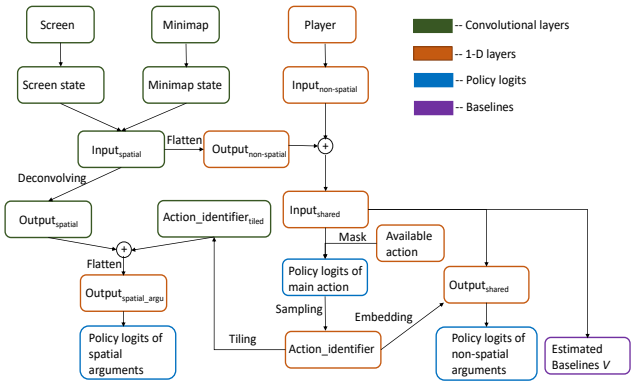

Fig. 3 Architecture of Neural Network Architecture

pected. However, in this work, we correct such mistakes, and results of our preliminary experiments have shown the strength of GA3C+SIL in sample efficiency in contrast with proposed GA3C.

3.3 Neural Network Architecture

Our neural network architecture is inspired from the one of *Control Agent* described in the study of RDRL [11], but has a smaller scale and a set of fewer parameters without *Memory processing* and *Relational processing*. The neural network architecture consists of three parts: *Input preprocessing*, *State encoding* and *Output processing*. Figure 3 sketches this neural network architecture.

Input prepocessing: the actor receives four kinds of information from SC2LE: *screen*, *minimap*, *player information* and *available actions*. These features are handled in the same way of RDRL. Here, we describe the way of logarithmic transformation and embedding in our work. Numerical features are transformed as $y = log(x + 1)$, where x is the value of a numerical feature. Categorical features are embedded into a continuous n-dimensional space, where $n = log_2(d)$[*1] and d represents the number of categories.

State encoding: *screen* and *minimap* are fed to two independent residual convolutional blocks [4]. The configuration of both residual convolutional blocks in this work is the same to those of study of RDRL. The output of blocks are concatenated along the dimension of output channels to form a spatial input, denoted as $input_{spatial}$. Output of *player information* is handled in the same way of the study of RDRL to form a non-spatial input, denoted as $input_{non\text{-}spatial}$.

Output processing: $input_{spatial}$ is flattened along the first two dimensions and passed to the neural network composed of two fully connected layers (512 units per layer, ReLU activations) to form a output, denoted as $output_{non\text{-}spatial}$. Then, $input_{non\text{-}spatial}$ and $output_{non\text{-}spatial}$ are concatenated along the last dimension to form a set of shared features, denoted as $input_{shared}$. In order to get policy logits of the action identifier and estimation of given state, $input_{shared}$ is passed to a fully connected layer (256 units, ReLU activation), followed by a fully connected layer ($|action\_identifier|$ units). Here, unavailable actions is masked according to *available action*. To get values of baseline V, we feed $input_{shared}$ to another neural network composed of two fully connected layers (256 units, ReLU activation, 1 unit).

Action identifier is sampled according to computed policy logits and embedded into a $log_2(|action\_identifier|)$ dimensional tensor, denoted as *action_identifier*. This tensor and $input_{shared}$ are concatenated along the last dimension to form a set of new shared features, denoted as $output_{shared}$. We use $output_{shared}$ to generate policy logits for all non-spatial arguments through independent neural networks made of one fully connected layer (one for each argument). For spatial arguments, we first tile the action identifier along dimension of output channels of $input_{spatial}$ to form a tiled output, denoted as $action\_identifier_{tiled}$. Then we deconvolve $input_{spatial}$ to a set of layers through two Conv2DTranspose layers (both 4 x 4 kernels, stride 2), whose output channels are 16 and 32 respectively. This output is denoted as $output_{spatial}$. $Output_{spatial}$ and $action\_identifier_{tiled}$ are concatenated along dimension of output channels, and passed to independent convolution layers (1 x 1 kernels, stride 1 and one for each spatial argument). Then, we flatten the output layers along the first two dimensions to form a output of spatial arguments, denoted as

[*1] https://github.com/simonmeister/pysc2-rl-agents

$output_{spatial\_argu}$. Finally, spatial arguments *(x, y)* are sampled from $output_{spatial\_argu}$.

4. Experiments and Results

We focus on all seven mini-games of StarCraft II provided by study of SC2, and train agents by proposed GA3C and GA3C+SIL described in Section 3.2. The names of these mini-games are *MoveToBeacon*, *CollectMineralShards*, *FindAndDefeatZerglings*, *DefeatRoaches*, *DefeatZerglingsAndBanelings*, *CollectMineralsAndGas* and *BuildMarines*. In this work, we take results from the study of SC2 as baselines for our experiments, and attempt to explore that how much SIL can contribute to improving sample efficiency in different mini-games. All introductions of seven mini-games are based on document[*2] offered by DeepMind. In this work, all the experiments are conducted for three times. All results shown in the following figures are the best one of three runs. We also provide all results of three runs in Table 1, where DHP denotes DeepMind Human Player, SGP denotes StarCraft GrandMaster Player, RP denotes Random Policy, FCL denotes FullConnected LSTM, and G denotes proposed GA3C, S denotes SIL.

Table 1 Results of all mini-games.
1-*MoveToBeacon*(Figure 4).
2-*CollectMineralShards*(Figure 5).
3-*FindAndDefeatZerglings*(Figure 6).
4-*DefeatZerglingsAndBanelings*(Figure 8).
5-*DefeatRoaches*(Figure 7).
6-*CollectMineralsAndGas*(Figure 9).
7-*BuildMarines* (Figure10)

Agent	1	2	3	4	5	6	7
DHP [10]	26	133	46	729	41	6880	138
SGP [10]	28	177	61	727	215	7566	133
RP [10]	1	17	4	23	1	12	< 1
FCL [10]	26	104	44	96	98	3351	6
G 1st	27	116	46	73	83	3349	1
G 2nd	27	117	47	89	94	3360	2
G 3rd	**27**	119	47	**89**	96	**3361**	4
G+S 1st	26	116	47	84	89	3323	11
G+S 2nd	27	124	47	86	97	3344	14
G+S 3rd	**27**	**136**	**49**	86	**110**	3353	**40**

4.1 General Configurations

In this work, most of the experiments are run with Nvidia GTX 2080Ti graphic cards, and the other experiments are run with Nvidia GTX 1080Ti graphic cards. Our implementation of GA3C is based on the open-sourced implementation of GA3C [*3], and our implementation of SIL is based on OpenAI Baselines [*4] and implementation of original SIL [*5]. Additionally, Sonnet library, which is built on top pf Tensorflow for convenience of building complex neural networks, is used in this work, and Tensorflow library of stable version 1.13.1 with cuda of version 10.0 is also used in this work. More details of hyperparameters and configuration can be found at Appendix 5.

Fig. 4 *MoveToBeacon* of proposed GA3C, GA3C+SIL

4.2 MoveToBeacon

In the beginning of this mini-game, one *marine* and one *beacon* is put on the map randomly. The agent is required to move the *marine* to the position of *beacon*(reward +1). When *marine* reaches current *beacon*, another *beacon* will be put on a different random position of the map. This process will be repeated until 120 seconds elapsed since the beginning of this mini-game.

Figure 4 shows the results of *MoveToBeacon*. For both GA3C and GA3C+SIL, the maximum mean score of 30 episodes are 27, which is larger than the one of study of SC2. Thanks to SIL, agents trained by GA3C+SIL learn faster than those trained by proposed GA3C.

4.3 CollectMineralShards

There are two *marines* on the center of the map and 20 *Mineral Shards* at the beginning of this mini-game. The goal of agent is to move the *marine* to collect the *Mineral Shards*, and rewards are earned (+1) when one mineral shard is collected. Whenever all 20 *Mineral Shards* have been collected, a new set of 20 *Mineral Shards* are spawned at random locations. The mini-game will be over until 120 seconds elapsed since the beginning of this mini-game.

Figure 5 shows the results of *CollectMineral-*

[*2] https://github.com/deepmind/pysc2/blob/master/docs/mini_games.md
[*3] https://github.com/NVlabs/GA3C
[*4] https://github.com/openai/baselines
[*5] https://github.com/junhyukoh/self-imitation-learning

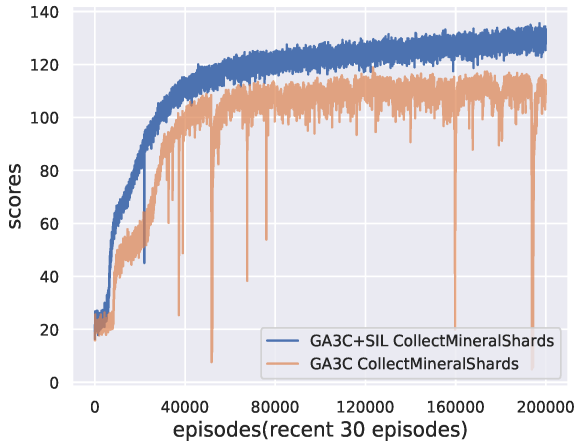

Fig. 5 *CollectMIneralShards* of proposed GA3C, GA3C+SIL

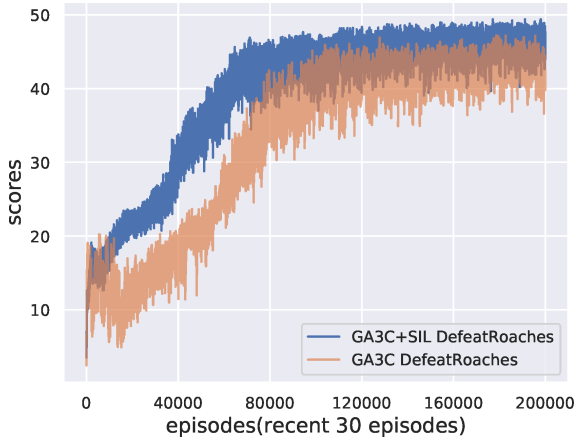

Fig. 6 *FindAndDefeatZerglings* of proposed GA3C, GA3C+SIL

Shards. The maximum mean score of 30 episodes for GA3C+SIL is 136, which is better than the one of GA3C. Thanks to SIL, the agents trained by GA3C+SIL learn faster and perform better than the agents trained by proposed GA3C through the whole training. We can also know the learning stability of GA3C+SIL is stronger than the one of proposed GA3C from figure 5. We think the better exploitation caused by SIL can result in more stable and deeper exploration.

4.4 FindAndDefeatZerglings

At the initial state of this mini-game, 3 *marines* are positioned in the center of the map and 25 *zerglings*(a kind of melee unit) are distributed randomly around the *marines*. Only in this mini-game the Fog-of-war is enabled to hide the information around the *marines*, so the agent is required to using *marines* to explore the map firstly, and then defeat the *zerglings* that are found during exploration. Rewards are earned when 1 *zerglings* is defeated (+1) or losing 1 *marine* (-1). When all 25 *zerglings* have been defeated, a new set of 25 *zerglings* are spawned at random positions in the map. The mini-game will be over if 120 seconds elapsed or all *marines* have been defeated.

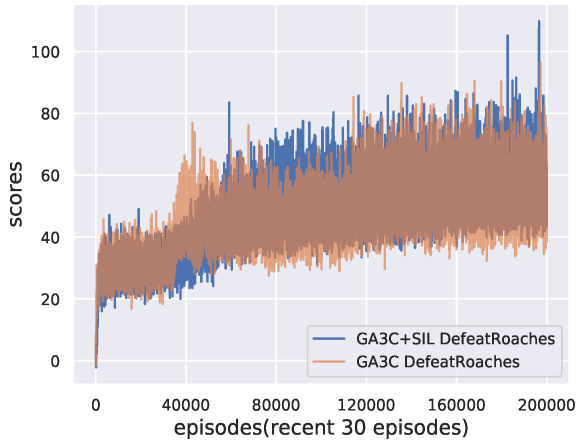

Fig. 7 *DefeatRoaches* of proposed GA3C, GA3C+SIL

Figure 6 shows the results of *FindAndDefeatZerglings*. The maximum mean score of GA3C+SIL is a little higher than the one of GA3C. Similar to the results of *MoveToBeacon*, thanks to SIL, the agents trained by GA3C+SIL learn faster than the agents trained by proposed GA3C at the early stage of training, and ultimately perform better.

4.5 DefeatRoaches

Initially, 9 *marines* and 4 *roaches* are positioned on opposite sides of the map. The agent is required to use *marines* to defeat as many *roaches* as possible, and rewards are earned when one roach is defeated (+10) or one marine is defeated (-1). Whenever all 4 *roaches* have been defeated, another 4 roaches are spawned and the agent is awarded 5 additional *marines* with other remaining *marines* retaining their health. Whenever new units are spawned, all unit positions are reset to opposite sides of the map. This mini-game can be over when 120 seconds elapsed or all *marines* are defeated. Figure 7 shows the results of *DefeatRoaches*. In contrast with the performance of *MoveToBeacon*, both agents of GA3C+SIL and proposed GA3C perform nearly to each other at the early stage of the training. However, SIL helps the agents trained by GA3C+SIL perform better than the those trained by proposed GA3C ultimately.

4.6 DefeatZerglingsAndBanelings

This mini-game will start with 9 *marines* on the opposite side from a group of 6 *zerglings* and 4 *banelings*. Since the *banelings* can cause dangerous area of effect (AOE) attack, and such splash damage is destructive to *marines* when they are bunched up, the agent is required to first defeat *banelings* and then defeat the remaining *zerglings* to gain more scores. Rewards are earned when one *baneling* or *zergling* is defeated (+5) or one marine is killed (-5). Whenever all *zerglings* and *banelings*

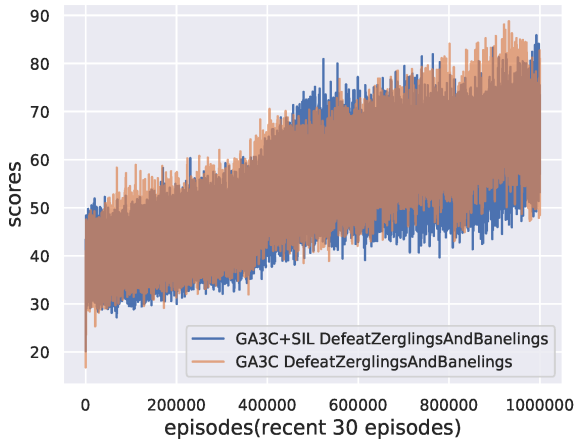

Fig. 8 *DefeatZerglingsAndBanelings* of proposed GA3C, GA3C+SIL

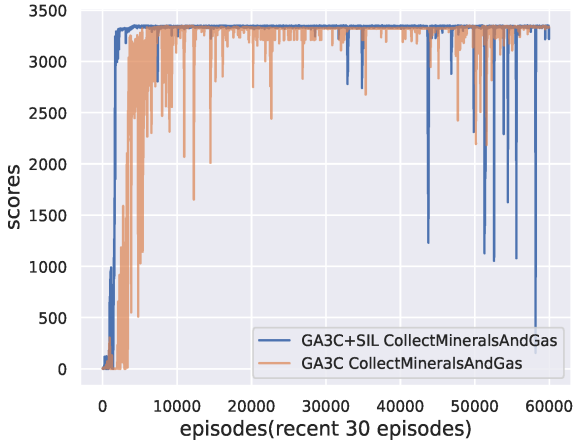

Fig. 9 *CollectMineralsAndGas* of proposed GA3C, GA3C+SIL

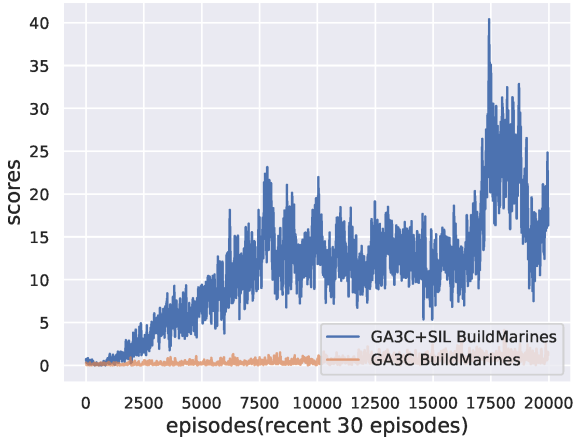

Fig. 10 *BuildMarines* of proposed GA3C, GA3C+SIL

have been defeated, a new group of 6 *zerglings* and 4 *banelings* is spawned, and the agent is awarded another 4 *marines* at full health, with all other surviving *marines* retaining their existing health. Whenever new units are spawned, all unit positions are reset to opposite sides of the map. This mini-game can be over when 120 seconds elapsed since the beginning of the mini-game or all *marines* are defeated.

Figure 8 shows the results of *DefeatZerglingsAndBanelings*. Unlike the performances of *DefeatRoaches*, the agents trained by GA3C+SIL in this mini-game perform a little worse than those trained by proposed GA3C. Since the reward is the same for both defeating a *zergling* and defeating a *baneling*, it can be a confusion for the agents to learn the optimal policy of defeating *banelings* firstly rather than defeating *zerglings* firstly. Additionally, since SIL depends a lot on the cumulative rewards of n-step playing to determine the advantage value of a certain state-action pair, reward configuration of *DefeatZerglingsAndBanelings* can also confuse the agents when distinguishing good experiences of defeating *banelings* from the other experiences.

4.7 CollectMineralsAndGas

In this mini-game, there are 12 SCVs, 1 Command Center, 16 Mineral Fields and 4 Vespene Geysers initially. The agent can use SCVs to collect minerals from Mineral Fields, and gather gas from Vespene Geysers. Rewards are based on the total amount of minerals and gas collected. Spending Minerals and Vespene Gas to train new units does not decrease your reward. The optimal policy of this mini-game require the agent construct additional SCVs to improve the efficiency of collecting minerals and gas to gain more rewards. The game can be over when 300 seconds elapsed since the beginning of the mini-game.

Figure 9 shows the results of *CollectMineralsAndGas*. At the early stage of the training, the agents trained by GA3C+SIL learn faster than those trained by proposed GA3C. Although both agents perform nearly to each other, there are some degrees of instabilities during both trainings. This is because in the original GA3C, the policy of DNN is potentially several updates ahead of the Agent's policy. Furthermore, the optimal policy of this mini-game is hard to learn because one inappropriate action caused by stochastic policy can result to very low score finally. We think better control over policy update can mitigate issues of instabilities in this mini-game.

4.8 BuildMarines

There are 12 SCVs, 1 Command Center, and 8 Mineral Fields initially in this mini-game. Rewards are earned by building *marines*. This is accomplished by using SCVs to collect minerals, which are used to build Supply Depots and Barracks, which can then build *marines*.

Figure 10 shows the results of *BuildMarines*. The agents trained by GA3C+SIL can find a way to obtain better exploration and get higher scores with training goes by. However, the agents trained by GA3C with the same amount of experiences can not obtain scores over

5. Such bad performance is similar to the one of study of SC2. They have trained the agents with much more experiences, but the scores can not even beyond 10. Therefore, we think that the optimal policy of this mini-game is hard to be found via A3C based methods only.

5. Conclusion

In five of seven mini-games (*MoveToBeacon*, *CollectMineralShards*, *FindAndDefeatZerglings*, *DefeatRoaches* and *CollectMineralsAndGas*), the results demonstrate that combined with GA3C, SIL indeed makes the learning and the training faster, and the trained agents achieve better performance with fewer experiences generated from SC2LE. Additionally, our simpler neural network shows its ability to capture the underlying informations in the observations better than that of study of SC2. We also hope this work can be inspiration of other work that focus on improving sample efficiency.

References

[1] Y. Aytar, T. Pfaff, D. Budden, T. Paine, Z. Wang, and N. de Freitas. Playing hard exploration games by watching youtube. In *Advances in Neural Information Processing Systems*, pages 2930–2941, 2018.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[3] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1407–1416, 2018.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Z. Hu and T. Kaneko. Reinforcement learning with effective exploitation of experiences on mini-games of starcraft ii. 2018.

[6] J. Oh, Y. Guo, S. Singh, and H. Lee. Self-imitation learning. In *International Conference on Machine Learning*, pages 3878–3887, 2018.

[7] OpenAI. Openai baselines: Acktr&a2c. https://openai.com/blog/baselines-acktr-a2c/.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations(ICLR)*, 2016.

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

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

[11] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019.

[12] W. Ziyu, B. Victor, H. Nicolas, M. Volodymyr, M. Remi, K. Koray, and F. Nando de. Sample efficient actor-critic with experience replay. In *International Conference on Learning Representations*, 2017.

Appendix

Table A·1 Hyperparameters of GA3C and GA3C+SIL

Hyper-parameters	Value
Learning rate	1e-4
Screen resolution	32x32
Minimap resolution	32x32
Number of Agents	32
Number of Predictor	4
Number of Trainer	8
Number of Recorder	8
Batch size	128
N-step	16
Baseline loss scaling	1.0
Entropy loss scaling	0.001
SIL update per iteration (M)	4
SIL batch size	512
Replay buffer size	150000
Exponent for prioritization	0.6
Bias correction for prioritized replay	0.4
Discount γ	0.99
Clip global gradient norm	100.0
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1e-8
Small positive constant ε	1e-6