Reinforcement Learning with Effective Exploitation of Experiences on Mini-Games of StarCraft II

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

Abstract: StarCraft II is a new challenge for the reinforcement learning with large state space and large action space. This paper reports empirical evaluation of the combination of GPU Asynchronous Advantage Actor-Critic (GA3C), residual neural networks and Self-Imitation Learning in mini-games of StarCraft II. The results show that GA3C with deeper architecture and residual neural networks achieved better performance than those reported in existing work.

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

1. Introduction

Deep reinforcement learning have achieved state-of-the-art performance on varieties of games, such as Atari games and game of Go [5]. StarCraft II is a real-time strategy game that has a lots of skilled human players, who can still defeat current computer programs. PySc2 [6] is an environment where researchers can make experiments on StarCraft II. Because the full game of StarCraft II is too difficult, mini-games are also provided in there. Each mini-game inherits a different difficulty in the full game. In the study of PySc2 [6], the performance of agents trained by Asynchronous Advantage Actor-Critic (A3C) is also published.

The main contribution of this paper is empirical evaluation of the combination of GPU Asynchronous Advantage Actor-Critic (GA3C), residual neural networks and Self-Imitation Learning in mini-games of StarCraft II. They are incorporated to improve the efficiency in learning. GA3C [1], which is an alternative to A3C, is supposed to increase training efficiency by utilizing GPU instead of CPU. We incorporate residual neural networks following recent studies on StarCraft II [7] or AlphaGo Zero. Self-Imitation learning [3] is a recent technique that enhances learning in domains with sparse reward.

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

2. Background

This section briefly reviews the environment of mini-games in StarCraft II and their difficulties in reinforcement learning. The difficulties mainly come from large observation and action space. Also, agents need to handle both spatial and non-spatial features (and/or actions).

2.1 Environment

To train agents for mini-games of StarCraft II, we used StarCraft II Learning Environment (SC2LE) environment. SC2LE was developed by both DeepMind and Blizzard Inc., which consists of three parts: Linux StarCraft II binary, the StarCraft II API and PySc2 [6]. Linux StarCraft II is a program of game itself only available for Linux system. StarCraft II API is an interface that provides full external control of StarCraft II and it is critical for creating scripted bots and replay analysis. PySc2 is a Python environment that wraps the StarCraft II API to ease the interaction between Python RL learning agents and StarCraft II, which means the agent can obtain observations from game environment, take action in the game, and gain scores via PySc2.

2.1.1 Observations

The observations consist of three different part; screen observation, minimap observation, and non-spatial observations. The first two parts are given as sets of feature layers each of which represents unit types, hit points, unit density, etc. While screen observation consists of 17 differ-

ent feature layers, minimap observation holds 7 feature layers. For full game, they provide a detailed view provided by a local camera and a coarse view of the entire world, respectively. However, the minimap and screen give a view of the same area with the same resolution, 32 2 , in our configuration of experiments in mini-games in this paper. Non-spatial observations give the amount of resources collected, the set of actions available, information about build queue, selected units, idle workers count, and last actions, etc.

2.1.2 Actions

Agents for StarCraft II need to handle many actions. A completely specified action a can be represented by an action-function identifier a_0 and a sequence of its arguments $(a_1, a_2 ... a_L)$ that a_0 requires. The number of arguments L depends on action identifier a_0. There are 524 different action-function identifiers with 13 possible types of arguments. The action space is large and actions fall under two categories: spatial and non-spatial. Spatial action needs spatial arguments $([(x_0, y_0)]$ or $[(x_0, y_0), (x_1, y_1)])$, which represents pixels on the screen or minimap where the action should be taken. Also, not all actions are available in each game step in StarCraft II. For example, if currently no unit is selected, then the commands related with unit are not available. These largeness and complexity of the action space contribute to the difficulty in learning.

The environment accepts one action per 8 game frames, which is approximately 180 Actions Per Minute (APM). The average human players achieves approximately 150 APM, while top players can reach 400 or more [6].

2.1.3 Rewards

For each mini-game in PySc2 [6], rewards are carefully designed so that an agent can receive positive or negative rewards with a reasonable frequency during an episode. We used these rewards in this paper.

The winning condition of a full game is to eliminate all of the opponent's buildings. To achieve the goal, an agent in PySc2 is required to handle many tasks; gathering sufficient resources, constructing buildings for upgrading and production of units, and strengthen his or her army. There are two different reward structures, ternary score 1 (win), 0 (tie), -1 (loss) or Blizzard score. While the ternary score is only given at the end of each episode, Blizzard score increases along with events during an episode, such as mining resources.

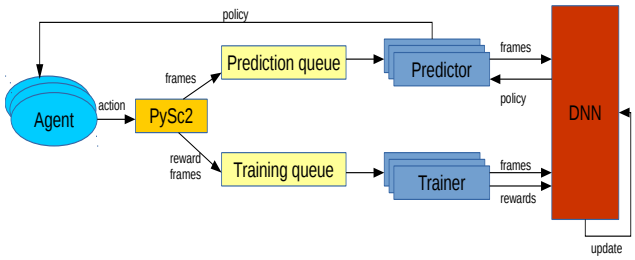

Fig. 1 Figure of Architecture of GA3C

2.2 A3C and GA3C

In the study of PySc2 [6], the performance of agents trained by Asynchronous Advantage Actor-Critic (A3C) is published. A3C is based on the Actor-Critic method, where the actor maintains and updates a policy and the critic maintains a value function or advantage function [2]. A3C deploys multiple actors running in parallel. The diversity of actors' experiences contribute to better exploration of different parts in a given environment, and also stabilize learning by reducing the correlation between samples.

GA3C extends A3C to utilize GPU [1]. In GA3C agents explore environments as actors do in A3C. To place our DNN in GPU apart from agents running on CPU, GA3C introduces two kinds of threads; Predictor and Trainer. Each predictor receives an observation of agents via prediction queue and returns the policy for the state stored in DNN (GPU). Each trainer receives a batch of experiences of the agents update DNN. Figure 1 sketches a basic mechanism of GA3C playing PySc2. GA3C keeps single, central neural network in GPU.

Both of actors in A3C and agents in GA3C interact with its own simulation environment and generate experiences, but agents do not keep their own copy of model and query the DNN for action policy. Value function is trained in GA3C as well, but all gradients are computed in GPU.

2.3 Self-Imitation Learning

Self-imitation learning improves the speed of learning and sample efficiency, by sampling better experiences more frequently. To exploit past good experience, a replay buffer similar to that in prioritized experience replay [4] is incorporated. The priority of each experience is given

by clipped advantage$(R - V_\theta(s))_+$, where $(.)_+$ denotes $max(., 0)$. The sampling probability of both of them is proportional to the attached priority. When combined with A3C, self-imitation learning is relatively simple to implement, as it does not involve importance sampling.

3. Proposed Method

This section describes our method. Compared to A3C agents reported with PySc2 [6], we replace A3C by GA3C [1], incorporate Self-Imitation Learning [3] and utilize deeper networks with a residual unit following recent study [7]. We believe GA3C is more suitable than A3C because StarCraft II game leads to high CPU utilization.

3.1 GA3C with Self-Imitation Learning

We build a combination of GA3C and Self-Imitation Learning, GA3C+SIL. Its architecture is shown in Figure 2. On one hand, the original Self-Imitation Learning (SIL) is built on A2C, though it is suggested that self-imitation learning can be combined with any actor-critic method [3]. Therefore, we need to extend the original SIL by asynchronous execution. On the other hand, the original GA3C does not have a replay buffer. Thus, we need to introduce a *Recorder* to receive the experience from agents and to store past experiences into our *replay buffer*. Then, *Trainer* receives on-policy samples from recorder queue and also sampled experiences from the *replay buffer*.

Algorithm 1 shows initialization of global data. Algorithm 2 and 3 show our agent and trainer, respectively. Note that multiple agents and trainers run in parallel asynchronously.In Algorithm 2, transition is denoted as (s_t, a_t, r_t), where s_t, a_t are a state and an action at time-step t, and $R_t = \Sigma_{k=t}^{\infty} \gamma^{k-t} r_k$ is the discount sum of rewards with discount factor γ. In Algorithm 3, π_θ, $V_\theta(s)$ represent policy and value function parameterized by θ respectively. H^π denotes entropy under the policy π and α is a weight for entropy regularization. L_{policy}^{ga3c}, L_{value}^{ga3c} are defined as policy loss and value loss when performing actor-critic via n-step samples respectively. Loss function L^{ga3c} is expectation of combination of policy loss and value loss under policy π_θ. β^{ga3c} is a weight for the value loss. Then, we perform self-imitation learning for M times. If the discount sum of rewards R is greater than value estimate V_θ, the agent learns to improve itself via this good experience. Otherwise, such a sampled experience is not used to update the parameter.

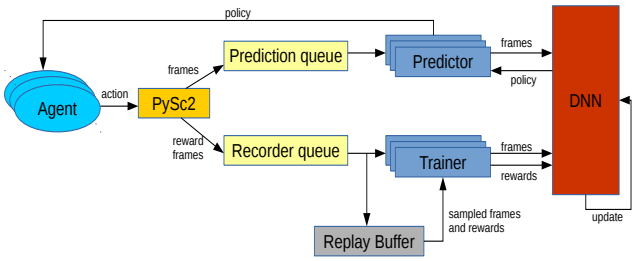

Fig. 2 Figure of Architecture of GA3C+SIL

Algorithm 1 GA3C+SIL: Initialize

```
Initialize parameter θ
Initialize shared step count T ← 0
Initialize replay buffer D ← φ
Initialize Recorder Queue Rq ← φ
Initialize Predict Queue Pq ← φ
Initialize Predictors {P_1, P_2, P_3...P_m}
```

Algorithm 2 GA3C+SIL: Agent

```
for all Agent ∈ {A_1, A_2, A_3...A_n} do in parallel
    episode buffer ε ← φ
    # Collects samples
    t_start ← T
    for each step do
        Enqueue s_t into Pq for all t in ε
        Obtain policy π_θ(a_t|s_t) via Pq
        Execute an action s_t, a_t, r_t, s_{t+1}~π_θ(a_t|s_t)
        Store transition ε ← ε ⋃ (s_t, a_t, r_t)
        T ← T + 1
        if s_{t+1} is terminal or T − t_start == t_max then
            # Update replay buffer
            Compute returns R_t = Σ_i^∞ γ^{i−t} r_i for all t in ε
            Partition ε into batches {b_1, b_2, b_3...b_q} of size N
            for all batch b ∈ {b_1, b_2, b_3...b_q} do
                Enqueue b into Rq
            end for
            Clear episode buffer ε ← φ
        end if
    end for
end parallel for
```

3.2 Network Architecture

Our network is deeper than that in study [6] and residual networks are incorporated is in study [7]. Figure 3 sketches the network architecture.

Input preprocessing: At every step, an agent observes four kinds of information: *screen*, *minimap*, *player*, and *available actions*. The features underlying these information can be divided into two kinds: *Categorical* and *Scalar*. Scalar features are adjusted with logarithmic transforma-

Algorithm 3 GA3C+SIL: Trainer

Initialize Trainers $\{Tr_1, Tr_2, Tr_3 ... Tr_p\}$
 # Perform actor-critic and self-imitation learning
for all each Trainer $\in \{Tr_1, Tr_2, Tr_3 ... Tr_p\}$ **do in parallel**
  # Get on-policy N-step batches from Recorder
  Dequeue batch b from Rq
  # Perform actor-critic using n-step samples
  for $(s_i, a_i, R_i) \in b$ **do**
    $D \leftarrow D \bigcup (s_i, a_i, R_i)$
    $H_i^{\pi} \leftarrow -\Sigma_a \pi(a|s_i) \log \pi(a|s_i)$
    $L_{policy}^{ga3c} \leftarrow -\log \pi_\theta(a_i|s_i)(R_i - V_\theta(s_i)) - \alpha H_i^{\pi}$
    $L_{value}^{ga3c} \leftarrow \frac{1}{2}\|R_i - V_\theta(s_i)\|^2$
  end for
  $L^{ga3c} \leftarrow \mathbb{E}_{s,a\sim\pi_\theta}[L_{policy}^{ga3c} + \beta^{ga3c} L_{value}^{ga3c}]$
  $\theta \leftarrow \theta - \eta \bigtriangledown_\theta L^{ga3c}$
  # Perform self-imitation learning using sampled experiences
  for $m = 1$ to M **do**
    Sample a mini-batch $\{(s, a, R)\}$ from D
    for $(s_k, a_k, R_k) \in \{(s, a, R)\}$ **do**
      if $R_k - V_\theta(s_k) > 0$ **then**
        $H_k^{\pi} \leftarrow -\Sigma_a \pi(a|s_k) \log \pi(a|s_k)$
        $L_{policy}^{sil} \leftarrow -\log \pi_\theta(a_k|s_k)(R_k - V_\theta(s_k)) - \alpha H_k^{\pi}$
        $L_{value}^{sil} \leftarrow \frac{1}{2}\|R_k - V_\theta(s_k)\|^2$
      end if
    end for
    $L^{sil} \leftarrow \mathbb{E}_{s,a,R\in D}[L_{policy}^{sil} + \beta^{sil} L_{value}^{sil}]$
    $\theta \leftarrow \theta - \eta \bigtriangledown_\theta L^{sil}$
  end for
end parallel for

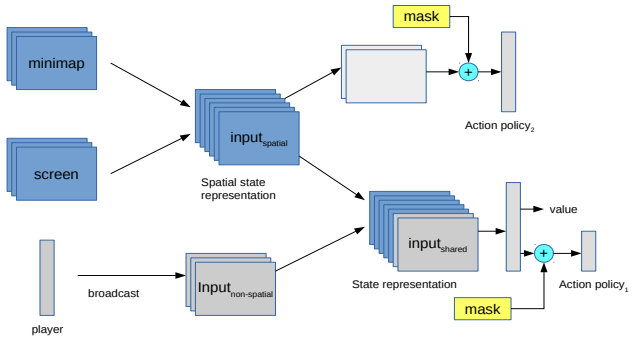

Fig. 3 Network Architecture

tion $x = \log(s)$, where s is value of scalar feature. Each categorical feature layer is embedded into a convolutional layer (y x y kernels with 1 filter) where $y = \log_2(d)$ and d is the dimension of the feature layer. The idea of $y = \log_2(d)$ is borrowed from an open source project `https://github.com/inoryy/pysc2-rl-agent`. While y is set to 10 according to input preprocessing in study of [7].

State encoding: *screen* and *minimap* are fed to independent residual convolutional blocks, each consists of 3 convolutional layer (5 x 5 kernels with 32 filters, 3 x 3 kernels with 48 filters, 3 x 3 kernels with 16 filters) followed by one residual block with 1 convolutional layer (1 x 1 kernels with 16 filters). All strides of convolutional layers here are set to 1. Then, these two outputs are concatenated along the depth dimension to represent spatial state input ($input_{spatial}$). The remaining input (*player*) is broadcast along the channel dimension of $input_{spatial}$, denoted as $input_{non\_spatial}$. Here, both dimensions of *screen* and *minimap* are the same but PySc2 allows us setting up different resolutions for these two inputs.

Table 1 Action Policy of Output

Action	Identifier a_0	Arguments $\{a_1, a_2 ... a_L\}$
Non-spatial action policy	Action policy$_1$	Action policy$_1$
Spatial action policy	Action policy$_1$	Action policy$_2$

Output processing: The network outputs two kinds of values: policy and value. They are computed differently for spatial actions and non-spatial actions.

The state representation is then formed by the concatenation of $input_{spatial}$ and $input_{non\_spatial}$, denoted as $input_{shared}$. Our network handles the identifier and arguments of spatial and non-spatial actions via Action policy1 and Action policy2, as summarized in Table 1. In order to get Action policy$_1$ and values of given state s, $input_{shared}$ is fed to a fully-connected layer with 256 units and ReLU activations, followed by fully-connected layer with $|actions|$ units and no activation. The mask of identifier a_0 is a 1-dimension list and only contains 1 and 0. The values of it are set to 1 whose index are obtained in *available action* otherwise 0. Mask of available arguments ($\{a_1, a_2 ... a_L\}$) of chosen action can be generated by the same manipulation as that of available actions. Then, Action policy$_2$ is obtained through 1 x 1 convolution of $input_{shared}$ with a single output channel and already generated mask. Finally, baseline value $V(s)$ is generated by passing $input_{shared}$ through a separated fully-connected layer with 256 units and ReLU activations, followed by fully-connected linear layer with 1 output unit.

4. Experiments

We compared the performance of GA3C and GA3C+SIL in *DefeatRoaches*, which is related with the goal of defeating as many *Roaches* as possible.

All hyper-parameters of experiments are described in Table 2. Our implementation of

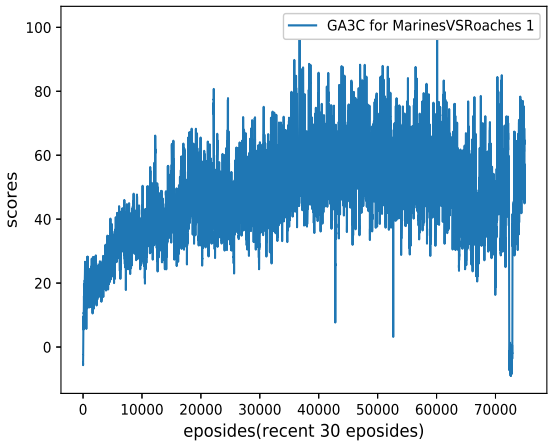

Fig. 4 Plain GA3C (1st run), the maximum average score (recent 30 eposides) was 101 across 75000 episodes.

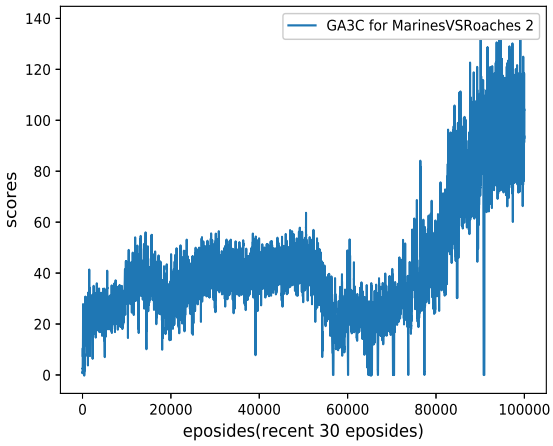

Fig. 5 Plain GA3C (2nd run): the maximum average score (recent 30 episodes) was 138 across 100000 episodes.

GA3C is based on the implementation by the authors of study [1], `https://github.com/NVlabs/GA3C`, and work of `https://github.com/inoryy/pysc2-rl-agent`. We used TensorFlow version 1.10 running on Linux with GeForce GTX 1080 and GeForce GTX 1080Ti, for most of our experiments.

There are totally seven mini-games in the SC2LE Environment. *DefeatRoaches* and *DefeatZerglingsAndBanelings* address the battle part of the full game among these mini-games. As *Banelings* is the only self-destructive military unit which causes damage to all enemies within a certain range, and *Roaches* and *Marines* are kinds of military unit that one unit can only attack one enemy within a certain range. Such units like *Marines* are common in the full game, so *DefeatZerglingsAndBanelings* can be considered as a special situation of battle part when compared to *DefeatRoaches*.

4.1 DefeatRoaches Task Description

The agent starts with nine *Marines* (one kind of military units that holds machine gun) on one side of the map and needs to defeat four *Roaches* on the other side of the map. Every time an agent defeats all the *Roaches*, it gets five more *Marines* at full health as reinforcement and four new *Roaches* spawn. The reward is +10 per *Roach* killed and -1 per *Marine* killed. The more *Marines* the agent keeps alive, the more *roaches* it can defeat. The end condition of this mini-game is either time up or all marines defeated. Additionally, game will be reset if four roaches were defeat every time.

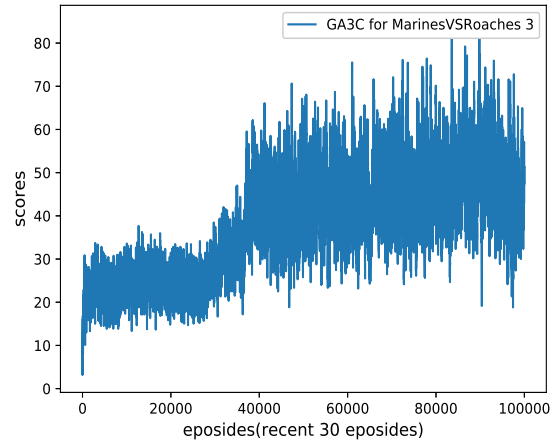

Fig. 6 Plain GA3C (3rd run): the maximum average score (recent 30 episodes) was 84 across 100000 episodes.

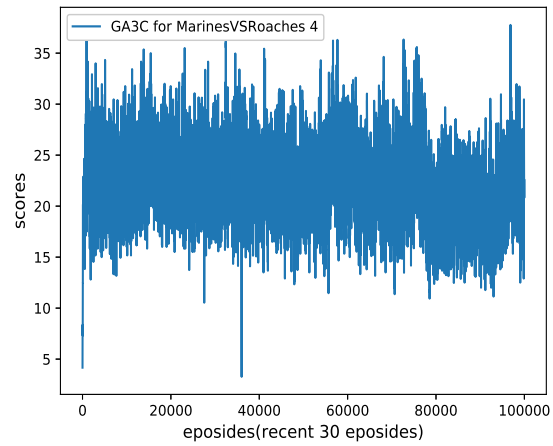

Fig. 7 Plain GA3C (4th run): the maximum average score (recent 30 episodes) was 38 across 100000 episodes.

4.2 Results

Figures 4, 5, 6, 7 are results of three different times of plain GA3C with proposed network

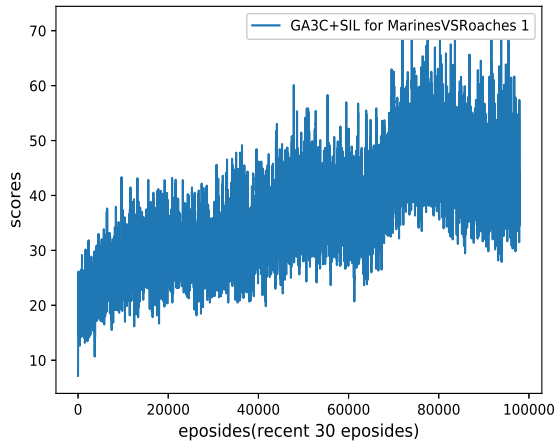

Fig. 8 GA3C+SIL (1st run): the maximum average score (recent 30 episodes) was 72 across 100000 episodes.

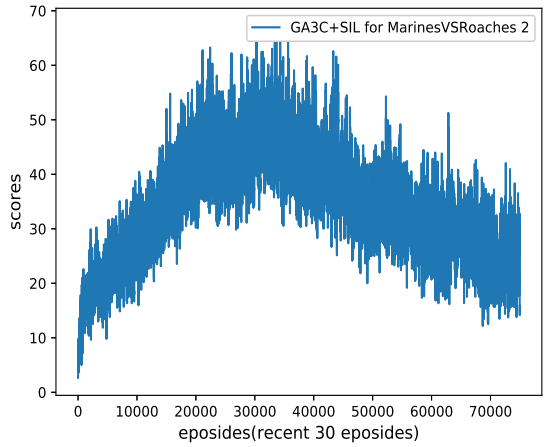

Fig. 9 GA3C+SIL (2nd run): the maximum average score (recent 30 episodes) was 67 across 100000 episodes.

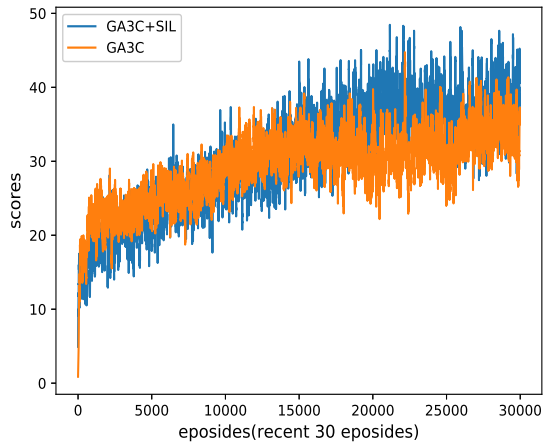

Fig. 10 Average of beginning stage of the training

architecture mentioned in Sect. 3.2. As shown in these three figures, the best average score of plain GA3C is over 100, which is the best average score reported in [6]. We believe our deeper network contributed to better performance. However, it is not always successful according to the result of 4th run, which is shown in 7. In the study of [6], it is also not always to achieve good performance of average score of 100. The reasons of failure may be the same as that of study [6]. Figure 8 and 9 show the results of GA3C+SIL. However, unexpectedly, the results of GA3C+SIL does not outperform those of plain GA3C. In Figure 9, we can see that the mean score decreases after 30000 episodes until the end of the training. Our speculation is that GA3C+SIL model may tend to overfit to the past experiences. In order to find out that whether SIL improves the beginning stage of learning or not, we take the average value of both plain GA3C and GA3C+SIL across 30000 episodes, as shown in Figure 10. As both agents performed similarly at the beginning of learning and agents of GA3C+SIL perform a little better than agents of plain GA3C, so SIL did not improve the performance at the beginning of learning a lot.

Table 2 Training Parameters Used

Hyper-parameters	Value
Learning rate	$4 * 10^{-4}$ annealing to $2 * 10^{-4}$ linearly
Screen resolution	32x32
Minimap resolution	32x32
Number of asynchronous agents (n)	16
Number of Predictor (m)	2
Number of Trainer (p)	4
Batch size (N)	32
Baseline loss scaling	0.5
Entropy regularization	0.001
SIL update per iteration (M)	4
SIL batch size	64
SIL loss weight	1
SIL value loss weight	0.02
Replay buffer size	10^5
Discount	0.99
RMSProp Decay	0.99
RMSProp Epsilon	$1 * 10^{-5}$

5. Conclusion

In this paper, we combined GA3C and self-imitation learning (SIL), and evaluated the method (GA3C+SIL) in DefeatRoaches, which is a mini-games of StarCraft II. Thanks to our deeper network with residual convolutional blocks, plain GA3C have shown better performance than existing work. However, the performance of GA3C+SIL was not better than that of GA3C.

6. Acknowledgement

A part of this work was supported by JSPS

KAKENHI Grant Number 16H02927 and by JST, PRESTO.

References

[1] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *International Conference on Learning Representations(ICLR)*, 2017.

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

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

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

[5] 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.

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

[7] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

