On Reinforcement Learning for Full-Length Game of StarCraft*

Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, Tong Lu

National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
pangzj@lamda.nju.edu.cn, liuruoze@163.com, misskanagi@gmail.com, zhangyi@smail.nju.edu.cn,
yuy@nju.edu.cn, lutong@nju.edu.cn

Abstract

StarCraft II poses a grand challenge for reinforcement learning. The main difficulties include huge state space, varying action space, long horizon, etc. In this paper, we investigate a set of techniques of reinforcement learning for the full-length game of StarCraft II. We investigate a hierarchical approach, where the hierarchy involves two levels of abstraction. One is the macro-actions extracted from expert's demonstration trajectories, which can reduce the action space in an order of magnitude yet remain effective. The other is a two-layer hierarchical architecture, which is modular and easy to scale. We also investigate a curriculum transfer learning approach that trains the agent from the simplest opponent to harder ones. On a 64×64 map and using restrictive units, we train the agent on a single machine with 4 GPUs and 48 CPU threads. We achieve a winning rate of more than 99% against the difficulty level-1 built-in AI. Through the curriculum transfer learning algorithm and a mixture of combat model, we can achieve over 93% winning rate against the most difficult non-cheating built-in AI (level-7) within days. We hope this study could shed some light on the future research of large-scale reinforcement learning.

Introduction

In recent years, reinforcement learning (Sutton and Barto 1998) (RL) has developed rapidly in many different domains. The game of Go has been considered conquered after AlphaGo (Silver et al. 2016) and AlphaGo Zero (Silver et al. 2017) . Many Atari games are nearly solved using DQN (Mnih et al. 2013) and follow-up methods. Various mechanical control problems, such as robotic arms (Levine et al. 2016) and self-driving vehicles (Shalev-Shwartz, Shammah, and Shashua 2016), have made great progress. However, reinforcement learning algorithms at present are still difficult to be used in large-scale reinforcement learning problems. Agents can not learn to solve problems as smartly and efficiently as human. In order to improve the ability of reinforcement learning, complex strategic games like StarCraft

have become the perfect simulation environments for many institutions such as DeepMind (Vinyals et al. 2017), FAIR (Tian et al. 2017), and Alibaba (Peng et al. 2017).

From the perspective of reinforcement learning, StarCraft poses a grand challenge. Firstly, it is an imperfect information game. Players can only see a small area of map through a local camera and there is a fog of war in the game. Secondly, the state space and action space of StarCraft are huge. StarCraft's image size is much larger than that of Go. There are hundreds of units and buildings, and each of them has unique operations, making the action space extremely large. Thirdly, a full-length game of StarCraft usually lasts from 30 minutes to more than an hour, and players need to make thousands of decisions to win. Finally, StarCraft is a multi-agent game. The combination of these issues makes StarCraft a great challenge for reinforcement learning.

Most previous agents in StarCraft are based on manual rules and scripts. Some works related to reinforcement learning are usually about micromanagement (e.g. (Usunier et al. 2016)) and macromanagement (e.g. (Justesen and Risi 2017)). These works solved some specific problems like local combat in StarCraft. However, there are rare works about the full-length games. In the paper of SC2LE (Vinyals et al. 2017), the benchmark result given by DeepMind shows that the A3C algorithm (Mnih et al. 2016) in SC2LE did not achieve one victory on the easiest level-1, which reveals the difficulty of the full-length game in StarCraft II. In the next section, we will have a summary of the difficulties encountered in the StarCraft II and introduce the simulation platform SC2LE (StarCraft II Learning Environment).

In this paper, we investigate a set of techniques of reinforcement learning for the full-length game of StarCraft II (SC2). In the section following the background, we present the hierarchical architecture investigated in this paper, which uses several levels of abstractions to make intractable large-scale reinforcement learning problems easier to handle. An effective training algorithm tailed to the architecture is also investigated. After that, we give some experiments in the full-length game on a 64x64 map of SC2LE. At last, we discuss about the impacts of the architecture, reward design and settings of curriculum learning. Experimental results achieved in several difficult levels of full-length games on SC2LE illustrate the effectiveness of our method. The main contributions of this paper are as follow:

- We investigate a hierarchical architecture which makes large-scale SC2 problem easier to handle.

- A simple yet effective training algorithm for this architecture is also presented.

- We study in detail the impact of different training settings on our architecture.

- Experiment results on SC2LE show that our method achieves state-of-the-art results.

Background

Reinforcement Learning

Consider a finite-horizon Markov Decision Process (MDP), which can be specified as 6-tuple:

$$M = \langle S, A, P(.), R(.), \gamma, T \rangle, \qquad (1)$$

S is the state space and $s \in S$ is a state of the state space. A is the action space and $a \in A$ is an action which agent can choose in state s. $P(.) = \Pr(s'|s, a)$ represents the probability distribution of next state s' over S when agent choose action a in state s. $R(.) = R(s, a)$ represents the instant reward gained from the environment when agent choose action a in state s. γ is discount factor which represents the influence of future reward on the choice at now. T is the max length of time horizon.

Policy π is a mapping or distribution form S to A. Assuming one agent, using a policy π, starts from state s_0, chooses an action a_0, gains a reward $r_0 = R(s_0, a_0)$, then moves to next state s_1 according to the distribution $\Pr(s_1|s_0, a_0)$ and repeats this process. This will generate a sequence τ below:

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, \ldots, \qquad (2)$$

There is a state s_{end} in which the agent will stop when the agent reaches this state. Process from s_0 to s_{end} is called one episode. The sequence τ in the episode is called a trajectory of the agent. For finite-horizon problem, when time step exceeds T, the exploration is also over. A typical RL algorithm requires thousands to millions of episodes to learn a policy. In one episode, discounted cumulative reward got by the agent is defined as:

$$G = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots, \qquad (3)$$

G is called return of the episode. A typical RL algorithm aims to find an optimal policy which maximizes the expected return.

$$\pi^* = \underset{\pi}{\arg\max} \, \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t) \right] \qquad (4)$$

Hierarchical Reinforcement Learning

When the dimension of the state space in the environment is huge, the space that needs to be explored exhibits exponential growth, which is called the curse of dimensionality problem in RL. Hierarchical reinforcement learning (HRL) solves this problem by decomposing a complex problem into several sub-problems and solving each sub-question in turn. There are some traditional HRL algorithms. Option (Sutton, Precup, and Singh 1999) made abstraction for actions.

MaxQ (Dietterich 1999) split up the problem by the value function decomposition. And ALISP (Andre and Russell 2002) provided a safe state abstraction method that maintains hierarchical optimality. Although these algorithms can better solve curse of dimensionality problems, they mostly need to be manually defined, which is time consuming and laborious. Another advantage of the HRL algorithm is that the resolution of the time is reduced, so that the problem of credit assignment over a long time scale can be better handled.

In recent years, some novel HRL algorithms have been proposed. Option-Critic (Bacon, Harb, and Precup 2017) is a method using theorem of gradient descent to learn the options and the corresponding policies simultaneously, which reduces the effort of manual designing options. However, the automatically learned options do not perform as well as non-hierarchical algorithms on certain tasks. FeUdalNetwork (Vezhnevets et al. 2017) designs a hierarchical architecture that includes a Manager module and a Worker module and proposes a gradient transfer strategy to learn the parameters of the Manager and Worker in an end-to-end manner. However, due to the complexity, this architecture is hard-to-tune. MLSH (Frans et al. 2017) proposes a hierarchical learning approach based on meta-learning, which enhances the learning ability of transferring to new tasks through sub-policies learned in multiple tasks. MLSH has achieved better results on some tasks than the PPO (Schulman et al. 2017) algorithm, but because its setting is multi-tasking, it is difficult to apply to our environment.

StarCraft II

Games are ideal environments for reinforcement learning research. RL problems on real-time strategy (RTS) games are far more difficult than the problems on Go due to complexity of states, diversity of actions, and long time horizon. Traditionally, researches on real-time strategy games are based on search and planning approaches (Ontañón et al. 2013). In recent years, some studies have used RL algorithms on RTS and one of the most famous RTS research environments is StarCraft. Previous works on StarCraft are mostly focused on local battles or part-length and often get features directly from game engine. (Usunier et al. 2016) presents a heuristic reinforcement learning algorithm combining exploration in the space of policy and back propagation. (Peng et al. 2017) introduces BiCNet based on multi-agent reinforcement learning combined with actor-critic. Although they have achieved good results, they are only effective for part-length game. ELF (Tian et al. 2017) provides a framework for efficient learning and a platform mini-RTS for reinforcement learning research. ELF also gives a baseline of A3C (Mnih et al. 2016) algorithm in a full-length game of mini-RTS. However, because the problem is relatively simple, there is still a great distance from the complexity of StarCraft.

SC2LE is a new research learning environment based on StarCraft II which is the follow-up of StarCraft. The location information for each unit is given in the engine of StarCraft. However, the location information of units and buildings need to be perceived from the spatial features in SC2LE.

Therefore, the spatial complexity of the state of its input is much larger than StarCraft I. At the same time, in order to simulate the real-hand movement of human, the action space in SC2LE is refined to each mouse click event, which greatly increases the difficulty of learning and searching. Benchmark result given by DeepMind shows that A3C algorithm (Mnih et al. 2016) did not achieve one victory on the easiest difficulty level-1, verifying the difficulties of full-length game in StarCraft II. In addition to full-length game, SC2LE also provides several mini-games for research. (Zambaldi et al. 2018) proposes a relation-based reinforcement learning algorithm, which achieved good results on these minigames. But the results on the full-length game are still not reported in this paper. Recently, there is a research (Sun et al. 2018) about the full-length game of StarCraft II. Our performance is close to their work, but we use less prior knowledge and effectively exploit spatial information. Moreover, we use less computing resources, which is a single machine with 4 GPUs and 48 CPU threads.

Methodology

In this section, we introduce our hierarchical architecture and the generation of macro-actions firstly. Then the training algorithm of the architecture is given. At last, we discuss the reward design and the curriculum learning setting used in our method.
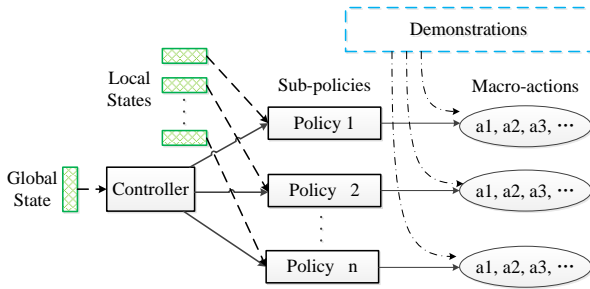

Figure 1: Overall Architecture.

Hierarchical Architecture

Our hierarchical architecture is illustrated in Fig. 1. There are two types of policies running in different timescales. The controller decides to choose a sub-policy based on current observation every long time interval, and the sub-policy picks a macro-action every short time interval.

For further illustration, we use Π to represent the controller. S_c and A_c is its state and action space. Similarly, assuming there is n sub-policies in the pool, we use $\langle \pi_1, \pi_2, ..., \pi_n \rangle$ to represent them. The state and action space of ith sub-policy is defined as S_i and A_i. R_i is its reward function. Besides, we have a time interval K. It means that the controller chooses a sub-policy in every K time units and the chosen sub-policy makes a decision in every time unit. Now we can deeply go through the whole process.

At time t_c, the controller gets its own global observation $s_{t_c}^c$, and it will choose a sub-policy i based on its state, like below:

$$a_{t_c}^c = \Pi(s_{t_c}^c), \quad s_{t_c}^c \in S_c \qquad (5)$$

Now the controller will wait for K time units and the ith sub-policy begins to make its move. We assume its current time is t_i and its local observation is $s_{t_i}^i$, so it gets the macro-action $a_{t_i}^i = \pi_i(s_{t_i}^i)$. After the ith sub-policy doing the macro-action $a_{t_i}^i$ in the game, it will get the reward $r_{t_i}^i = R_i(s_{t_i}^i, a_{t_i}^i)$ and its next local observation $s_{t_i+1}^i$. The tuple $(s_{t_i}^i, a_{t_i}^i, r_{t_i}^i, s_{t_i+1}^i)$ will be stored in its local buffer D_i for the future training. After K moves, it will return to the controller and wait for the next chance.

The high level controller gets the return of the chosen subpolicy π_i and compute the reward of its action $a_{t_c}^c$ as follows:

$$r_{t_c}^c = r_{t_i}^i + r_{t_i+1}^i + ... + r_{t_i+K-1}^i \qquad (6)$$

Also, the controller will get its next global state $s_{t_c+1}^c$ and the tuple $(s_{t_c}^c, a_{t_c}^c, r_{t_c}^c, s_{t_c+1}^c)$ will be stored in its local buffer D_c. Now the time is $t_c + 1$, and the controller will make a next choice based on its current global observation.

Therefore, we can see that there are some advantages in our hierarchical architecture. Firstly, each sub-policy and the high-level controller have different state space. The controller only needs the global information to make high-level decision. The global state space S_c is a small part of all the state space S. Also, a sub-policy responsible for combat is more focused on local state space S_i related to battle. It can be seen that such a hierarchical structure can split the original huge state space into a plurality of subspaces corresponding to different policy networks. Secondly, the hierarchical structure can also split the tremendous action space A. The sub-policies with different functions will have their own action space A_i. Thirdly, the hierarchical architecture can effectively reduce the execution step size of the strategy. Since the control network calls a sub-network every fixed time interval K, the total execution step size of the high-level network becomes T/K steps. The execution step size of sub-policies will also be reduced. Last but not least, the hierarchical architecture makes the design of the reward functions easier. Different sub-policies may have different targets. Therefore, they can learn more quickly by their own suitable reward functions.

Generation of Macro-actions

In StarCraft, the original action space A is tremendous. And human players always need to do a sequence of raw actions to achieve one simple purpose. For example, if we want to build a building in the game, we have to select a worker, order it to build the building in the specific position, and make it come back after finish. The sequences of the raw actions for some simple purposes are more likely some fixed sequence stored in mind for our human players. So we instead generate a macro-action space A^η which is obtained through data mining from trajectories of experts. The original action space A is then replaced by the macro-action space A^η. This will improve the learning efficiency and testing speed. The generation process of macro-actions is as follow:

- Firstly, we collect some expert trajectories which are sequence of operations $a \in A$ from game replays.

- Secondly, we use a prefix-span (Yan, Han, and Afshar 2003) algorithm to mine the relationship of the each operation and combine the related operations to be a sequence of actions a^{seq} of which max length is C and constructed a set A^{seq} which is defined as $A^{seq} = \{ a^{seq} = (a_0, a_1, a_2, ...a_i) \mid a_i \in A$ and $i \leq C, \}$

- Thirdly, we sort this set by $\mathrm{frequency}(a^{seq})$.

- Fourthly, we remove duplicated and meaningless ones, remaining the top K ones. Meaningless refers to the sequences like continuous selection or camera's movement.

- Finally, the reduced set is marked as newly generated macro-action space A^η.

Using the macro-action space A^η, our MDP problem is now reduced to a simple one, which is defined as:

$$M = \langle S, A^\eta, P(.), R(.), \gamma, T \rangle, \tag{7}$$

Meanwhile, the MDP problem of each sub-policy is also reduced to new one:

$$M_i = \langle S_i, A_i^\eta, P(.), R_i(.), \gamma, T_i \rangle, \text{ for } i = 1 \text{ to } n, \tag{8}$$

Training Algorithm

The training process of our architecture is showed in Algorithm 1 and can be summarized as follows. Firstly we initialize the controller and sub-policies. Then we run the iteration Z times and run the episode M times in each iteration. At the beginning of each iteration, we will clear all the replay buffers. In each episode, we collect the trajectories of the controller and sub-policies. At the end of each iteration, we use their own buffers to update the parameters of the controller and sub-policies.

The update algorithm we use is PPO (Schulman et al. 2017). Entropy's loss was added to the PPO's loss calculation to encourage exploration. Therefore, our loss formula is as follows:

$$L_t(\theta) = \hat{\mathbb{E}}_t[L_t^{\mathrm{clip}}(\theta) + c_1 L_t^{\mathrm{vf}}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{9}$$

where c_1, c_2 are the coefficients we need to tune, and S denotes an entropy bonus. $L_t^{\mathrm{clip}}(\theta)$ is defined as follows:

$$L_t^{\mathrm{clip}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{10}$$

$$L_t^{\mathrm{vf}}(\theta) = \hat{\mathbb{E}}_t[(r(s_t, a_t) + \hat{V}_t(s_t) - \hat{V}_t(s_{t+1}))^2] \tag{11}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, $\hat{A}_t$ is computed by a truncated version of generalized advantage estimation.

Reward Design

There are three types of rewards we explore in this paper. Win/Loss reward is a ternary 1 (win) / 0 (tie) / -1 (loss) received at the end of a game. Score reward is Blizzard score get from the game engine. Mixture reward is our designed reward function.

It's hard to learn the game using Win/Loss reward. Blizzard scores can be seen as dense reward. We will show that

---

Algorithm 1 HRL training algorithm

Input: Number of sub-policies N, time interval K, reward function $R_1, R_2, ..., R_n$, max episodes M, max iteration steps Z, max game steps T,

  Initialize replay buffer $\langle D_c, D_1, D_2, ..., D_n \rangle$, controller policy Π_ϕ, each sub-policy π_{θ_i}
  for $z = 1$ to Z **do**
    clear $\langle D_c, D_1, D_2, ..., D_n \rangle$
    for $m = 1$ to M **do**
      clear $\langle \tau_c, \tau_1, \tau_2, ..., \tau_n \rangle$
      for $t = 0$ to T **do**
        if t mod $K == 0$ **then**
          if $s_{t_c-1}^c$ is not None **then**
            $\tau_c \leftarrow \tau_c \cup \{(s_{t_c-1}^c, a_{t_c-1}^c, r_{t_c-1}^c, s_{t_c}^c)\}$
          end if
          $a_{t_c}^c \leftarrow \Pi_\phi(s_{t_c}^c), \quad r_{t_c}^c \leftarrow 0$
          $j \leftarrow a_{t_c}^c$
        end if
        collect experience by using π_{θ_j}
        $\tau_j \leftarrow \tau_j \cup \{(s_t^j, a_t^j, R_j(s_t^j, a_t^j), s_{t+1}^j\}$
        $r_{t_c}^c \leftarrow R_j(s_t^j, a_t^j) + r_{t_c}^c$
      end for
      $D_c \leftarrow D_c \cup \tau_c$
      for $i = 1$ to N **do**
        $D_i \leftarrow D_i \cup \tau_i$
      end for
    end for
    using D_c to update ϕ to maximize expected return
    for $i = 1$ to N **do**
      using D_i to update θ_i to maximize expected return
    end for
  end for

---

using this score as a reward can not help agent get more chances to win in the experiment section. We have designed reward functions for the sub-policies which combine dense reward and sparse reward. These rewards seem to be really effective for training, which will be shown in the experiment section.

Curriculum Learning

SC2 includes 10 difficult levels of built-in AI which are all crafted by rules and scripts. From level-1 to level-10, the built-in AI's ability is constantly improved. Training on higher difficulty levels gives less positive feedback, making it difficult for agent to learn from scratch. In this work, we trained our agent on lower difficulty levels at first, then transferred the agent to higher difficulty levels using the pre-trained model as the initial model, following the idea of curriculum learning.

However, when the pre-trained agent transfers to high difficulty levels, we find that if controller and all sub-policies are still updated at the same time, the training is sometimes unstable due to the mutual influence of different networks. In response to this situation, we have devised a strategy to update the controller and sub-policies alternatively. We found that this method can make the training more stable, and the

winning rate for high difficulty levels can rise steadily.

Experiments

In this section, we present the experiment results on SC2LE. We first introduce the experiment settings including the hierarchical architecture and three combat models. Then the details of experiments are shown. If you want to get more information about experiment settings and implementation details such as features of state space or forms of macro actions, please refer to the appendix of our arxiv version [1].

Setting

The setting of our architecture is as follow: controller selects one sub-policy every 8 seconds, and the sub-policy performs macro-actions every 1 second. In the setup, we have two sub-policies in the sub-policy pool. One sub-policy controls the construction of buildings and the production of units in the base, called base network. The other sub-policy is responsible for battle, called battle policy. This sub-policy has three different models, which are explained later.

A full-length game of SC2 in the 1v1 mode is as follow: First, two players spawn on different random points in the map and accumulate resources. Second, they construct buildings and produce units. At last, they attack and destroy each other's units and buildings. Fig. 2 shows a screenshot of the running game.

Figure 2: Screenshot of StarCraft II.

For simplicity, we set our agent's race to *Protoss* and build-in AI's race to *Terran*. But our archietecture could generalize to any race against any other race. The map we used is the 64x64 map *simple64* on SC2LE. We set the maximum length of each game to 15 minutes. Moreover, our agent does not open sub-mine, and only uses the two basic military units which are *Zealot* and *Stalker*.

In our setup, the battle policy has three different settings. These are explained below.

Combat rule   Combat rule is a simple battle policy, and there is only one action in combat rule model: attacking a sequence of fixed positions. Although the map is unobservable and the enemy's position is unknown, the enemy always lives around the mines. We only need to make our army attack the fixed positions around the mines. The attack action uses the built-in AI to do automatic move and attack. Thus the result of the attack action depends on the construction of the buildings and the production of the units. Only when the agent learns to better carry out building construction (for example, do not build redundant buildings, and build more pylons in time when supply is insufficient), it is possible for the agent to win.

Combat network   Though the simple combat rule model is effective, the combat process is slightly naive or rigid and may fail when moving to larger and more complex maps. Below we introduce a smarter attack approach which is called combat network. The output of the combat network consists of three actions and a position vector. These actions are: all attack a certain position, all retreat to a certain position, do nothing. The positions of attack and movement are specified by the position vector.

The combat network is constructed as a Convolution Neural Network (CNN). This CNN accepts feature maps of minimaps and screens which enable it to know the information of the full map and the positions of the units and buildings on the screen. Moreover, we use a simple strategy to decide the position of the camera. When the controller chooses the base sub-policy, the camera is moved to the location of agent's base. When the controller chooses the battle sub-policy, the camera is moved to the location of army. The location of army can be chosen in two ways. The first is the center point of all combat units. The second is the center of the most injured unit. Since the injured unit indicated the occurrence of a recent battle, we found that the last setting was better in practice.

Mixture model   Although the combat network model can be trained well on high levels of difficulty, it will occasionally miss some hidden enemy buildings. We can combine combat network with combat rule into a mixture model. When a certain value is predicted in the position vector of the combat network, the army's attack position will become a series of fixed positions got by prior knowledge.

Comparison of Training Method

In the following, we will discuss our training processes from three aspects. Firstly, curriculum learning is the major way we use to train our agents against the enemy from low difficulty levels to high difficulty levels. Secondly, our hierarchical architecture can be used for module training, which is convenient for replacing or refining some sub-policies while others fixed. Finally, our simultaneous training algorithm is sometimes unstable on high difficulty levels, so we instead use an alternative update strategy.

Effect of curriculum learning   In all of our combat models, we firstly train them on low difficulty levels and then transfer them to high difficulty levels. This process follows
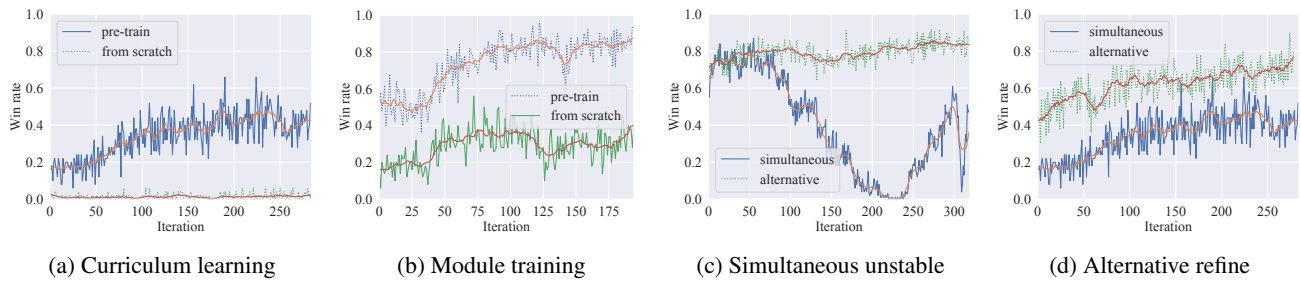
| (a) Curriculum learning | (b) Module training | (c) Simultaneous unstable | (d) Alternative refine |

Figure 3: Winning curve in training process.

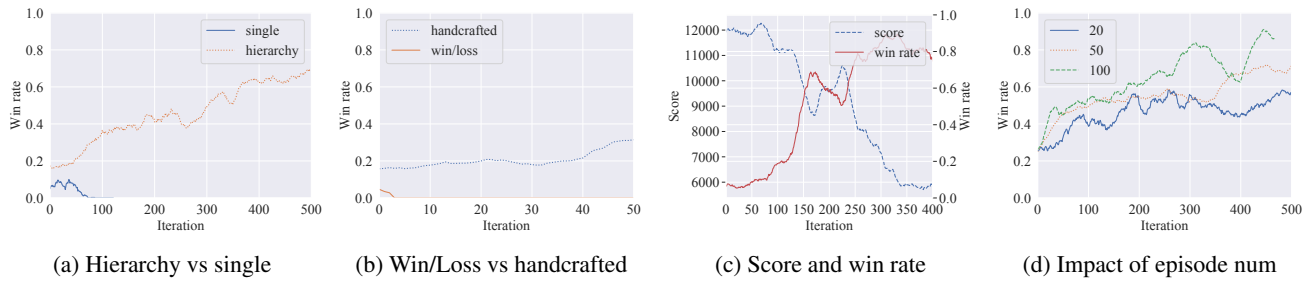| (a) Hierarchy vs single | (b) Win/Loss vs handcrafted | (c) Score and win rate | (d) Impact of episode num |

Figure 4: Comparison of settings.

the way of curriculum learning. We found that when training directly on high difficulty levels, the performance of our agent is difficult to improve. If we start training from low difficulty level and then use the low-difficulty's model as the initial model to update, we could often get better results.

Fig. 3 (a) shows the comparison between training the pre-trained agent and training from scratch on difficulty level-7. These two agents are using combat network model, and the first agent has been trained on difficulty level-2 and difficulty level-5.

Effect of module training   Since we are using a hierarchical architecture, we can easily replace sub-policies. While replacing a sub-policy, the parameters of the controller network and other sub-policies are retained, only the parameters of the newly replaced sub-policy are updated, which can accelerate learning. In the experiment, we have three types of combat strategies. If we have trained the simple combat rule model on high difficulty levels, we can use the parameters of its controller network and base network for the other two combat models.

Fig. 3 (b) shows the comparison between training using pre-trained networks and training from scratch on difficulty level-2. These two agents are using combat network model, and the first agent uses the parameters of controller and base network which are from the combat rule model trained on difficulty level-7.

Effect of simultaneous updating   For the sample efficiency, in our training algorithm, we let all the networks of the hierarchical architecture collect their own data and then update at the same time. They are sometimes unstable. Because every network would see the other networks as the part of environment, and do not consider the changes of the

others while updating. These unstable training rarely happened on lower difficulty levels, but can be seen on higher difficulty levels. We can use an alternate updating strategy to train the networks which can result in a steady improvement, but need more samples. We can also make the learning rate smaller which could also alleviate this problem. But the training process would be slower and sometimes hard to be improved.

Fig. 3 (c) shows the comparison between simultaneously updating and alternately updating on difficulty level-7. The agent uses combat rule model, and has been trained on difficulty level-2 and difficulty level-5.

Fig. 3 (d) also shows the comparison between simultaneously updating and alternately updating on difficulty level-7, but there are some differences. The agent uses combat network model and has been trained on difficulty level-2 and difficulty level-5. The learning rate is half smaller than Fig. 3 (c). The green line above is actually loading the trained model of the blue line below. In the experiment, we found that the blue line below could not be improved in the following training period, but the green line above could quickly improve after loading its model.

Comparison of Combat Models

After training the three combat models on difficulty level-7, we perform evaluation for each of them. Each evaluation tests from difficulty level-1 to difficulty level-10. We run 100 games in each difficulty level and report the winning rate. The result is shown in Table 1. In the difficulty level-1 to level-7, we found that the training agents have a very good performance. The built-in AI in difficulty level-8 to level-10 uses several different cheat techniques and select different strategies randomly. So the performances of them

Table 1: Evaluation Results.

Opponent's Type	Non-cheating (Training)							Cheating (No-training)		
Difficulty Level	1	2	3	4	5	6	7	8	9	10
Combat Rule	1	0.99	0.94	0.99	0.95	0.88	0.78	0.70	0.73	0.60
Combat Network	0.98	0.99	0.45	0.47	0.39	0.73	0.66	0.56	0.52	0.41
Mixture Model	1	1	0.99	0.97	1	0.90	0.93	0.74	0.71	0.43

are unstable and with a bit of randomness. However, it can be seen that our trained agents still have a good performance on fighting against them. The videos of our agents can be searched by "NJU-SC2-Bot" on Youtube.

Combat rule   The combat rule agent achieves good results in difficulty level-1 to difficulty level-5. We find that since agents can only use the two basic military units to fight, they are more inclined to use a fast attack fashion (called 'Rush'), which can guarantee the winning rate.

The behaviors of our learned agent are as follow. At the beginning, the agent tends to produce more workers to develop the economy. When the number of workers is close to saturation, the agent will turn to produce soldiers. When the number of soldiers is sufficient, the agent will choose to attack. This layered progressive strategy is automatically learned through our reinforcement learning algorithm which illustrates the effectiveness of our approach.

Combat network   We also test the combat network model on all 10 difficulty levels. We find that although the combat network model achieves a good winning rate on difficulty level-7, the winning rates on several other difficulty levels are not high. It is worth noting that many results of the other levels of difficulty are tie. This means that the model of the combat network is more difficult to completely destroy the enemy (which means eliminating all enemy buildings). Some enemy's buildings or units may be hidden somewhere. If we also see tie as victory, the winning rate of combat network is still high.

Mixture model   It can be found in Table 1 that the mixture model of combat network and combat rule has achieved best results in difficulty level 1-9. This can be explained as follow. The agent can not only choose the attacking area within the camera, but also can switch to a fierce attack on some certain points. This freedom can lead to a performance improvement which makes mixture model best in all three combat models.

Comparison of Settings

In this section we have three experiments to show the importance of hierarchy, design of reward and impact of hyper parameters.

Hierarchy vs non-hierarchy   As we use a hierarchical architecture, there is a common question which is that whether the SC2 problem can be handled by using a non-hierarchical architecture. In the paper of SC2LE, the learning effect of the original action and state space and non-hierarchical reinforcement learning algorithm has been given. It can be

seen that the performance was not satisfied. Our architecture has two levels of abstraction. One is a reduction in the action space, which is done by macro-action. The other is a two-layer architecture that uses the controller to select sub-policies. We tested the effect of keeping the use of macro-actions without using the controller and sub-policies. This is called the single-policy architecture.

We were surprised to find that on low difficulty levels, the single-policy architecture can be almost as good as hierarchical architecture learning which means that macro-actions are effective for training a good agent. However, when moving to high level of difficulty, the final performance of the hierarchical architecture are significantly better than the single-policy architecture, as shown in Fig. 4 (a), which is trained on difficulty level-7. It can be explained that when the difficulty level is low, the difference between the hierarchical and non-hierarchical architecture is less obvious. When the difficulty of the problem continues to increase, the performance of the hierarchical model will be better than the performance of non-hierarchical model. Another reason for using hierarchy is modularity. Modularity facilitates the replacement of sub-policy. For example, if we need to replace battle sub-policy, we can still retain the parameters of other networks which speeds up training.

Outcome reward vs handcrafted reward   Win/Loss reward can achieve good results on low difficulty level like handcrafted reward. However, when training is on high difficulty level, we found that the performance of Win/Loss reward on hierarchical model is relatively poor as shown in Fig. 4 (b) in which the agent is using combat network model and trained on difficulty level-7.

In SC2LE, it is mentioned that one can use Blizzard score as a reward. However, we found that Blizzard score and winning rate are sometimes not in a proportional relationship, especially on low difficulty levels. While the agent is trying to improve the score, it may ignore the attack chance on the enemy base and lose the best opportunity to win. This is shown in Fig. 4 (c) in which the agent is using combat network model and trained on difficulty level-2.

Influence of hyper-parameters   We have experimented with a variety of different parameters and found that the number of episodes in each iteration has some influence on the learning speed of the agent. When the number is small, the learning will be unstable and hard to converge. Improving the number of episodes can mitigate the problem. This effect is shown in Fig. 4 (d) in which the agent is using combat rule model and trained on difficulty level-2. Other parameters have little effect on training.

Conclusion

In this paper, we investigate a set of techniques of reinforcement learning for the full-length games in StarCraft II, including hierarchical policy training with extracted macro-actions, reward design, and curriculum design. With limited computation resource, we show that the combined approach achieves the state-of-the-art results beating the built-in AI. In the future, we will continue to explore more RL algorithms that can better solve large-scale problems.

References

Andre, D., and Russell, S. J. 2002. State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, 119–125.

Bacon, P.; Harb, J.; and Precup, D. 2017. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 1726–1734.

Dietterich, T. G. 1999. Hierarchical reinforcement learning with the MAXQ value function decomposition. *CoRR* cs.LG/9905014.

Frans, K.; Ho, J.; Chen, X.; Abbeel, P.; and Schulman, J. 2017. Meta learning shared hierarchies. *CoRR* abs/1710.09767.

Justesen, N., and Risi, S. 2017. Learning macromanagement in starcraft from replays using deep learning. 162–169.

Levine, S.; Finn, C.; Darrell, T.; and Abbeel, P. 2016. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research* 17:39:1–39:40.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing atari with deep reinforcement learning. *CoRR* abs/1312.5602.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 1928–1937.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Trans. Comput. Intellig. and AI in Games* 5(4):293–311.

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

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *CoRR* abs/1707.06347.

Shalev-Shwartz, S.; Shammah, S.; and Shashua, A. 2016. Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR* abs/1610.03295.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T. P.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

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

Sun, P.; Sun, X.; Han, L.; Xiong, J.; Wang, Q.; Li, B.; Zheng, Y.; Liu, J.; Liu, Y.; Liu, H.; and Zhang, T. 2018. Tstarbots: Defeating the cheating level builtin AI in starcraft II in the full game. *CoRR* abs/1809.07193.

Sutton, R., and Barto, A. 1998. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.* 112(1-2):181–211.

Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. ELF: an extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 2656–2666.

Usunier, N.; Synnaeve, G.; Lin, Z.; and Chintala, S. 2016. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *CoRR* abs/1609.02993.

Vezhnevets, A. S.; Osindero, S.; Schaul, T.; Heess, N.; Jaderberg, M.; Silver, D.; and Kavukcuoglu, K. 2017. Feudal networks for hierarchical reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 3540–3549.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; Quan, J.; Gaffney, S.; Petersen, S.; Simonyan, K.; Schaul, T.; van Hasselt, H.; Silver, D.; Lillicrap, T. P.; Calderone, K.; Keet, P.; Brunasso, A.; Lawrence, D.; Ekermo, A.; Repp, J.; and Tsing, R. 2017. Starcraft II: A new challenge for reinforcement learning. *CoRR* abs/1708.04782.

Yan, X.; Han, J.; and Afshar, R. 2003. Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*, 166–177.

Zambaldi, V. F.; Raposo, D.; Santoro, A.; Bapst, V.; Li, Y.; Babuschkin, I.; Tuyls, K.; Reichert, D. P.; Lillicrap, T. P.; Lockhart, E.; Shanahan, M.; Langston, V.; Pascanu, R.; Botvinick, M.; Vinyals, O.; and Battaglia, P. 2018. Relational deep reinforcement learning. *CoRR* abs/1806.01830.