Application of Deep Reinforcement Learning in StarCraft II Learning Environment

スタークラフトII学習環境における
深層強化学習の応用

Yang Xu

Supervisor: 鶴岡慶雅准教授

Graduate School of Engineering
University of Tokyo

This thesis is submitted for the degree of
Master of Engineering

Tsuruoka Lab February 2018

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Yang Xu
February 2018

Acknowledgements

I would like to first thank my parents for their support during these two years. You have helped me both financially and mentally. Without your help, I won't be able to go through these alone. It is the luckiest thing in world to have you.

A lot of appreciation must be given to my mentor professor Tsuruoka. Who has gave me this chance to carry out research and trusted me in these two years. Who also allowed me to research on my interested topics freely. Professor Tsuruoka provided support to me whenever it is required.

Also, I would like to thank every member of the Tsuruoka Lab. Who has helped me to live in Japan. Special thanks would be given to the lads in the same year with me. Your dedication to life, work and research affected me a lot.

Although it is very likely I won't be heard. I would like to thank the engineers working at DeepMind and Blizzard Entertainment for releasing the StarCraft II Learning Environment in 2017. So I have the software support to carry out this research.

Last but not least, I would like to give special thanks to my beloved Ni Yi. You have helped me at some of my lowest moments. You have shown me wonderful things in the world I would never notice without you.

Abstract

Using games to test and display the capabilities of Artificial Intelligence have been successful and popular. Dating back to 1959, A. Samuel has started to work on AI with computer checkers. In 1997, Deep Blue defeated the human world champion in chess. More recently, AlphaGo and its subsidiaries have defeated human world champion players in the game of Go using deep reinforcement learning based methods. After the successes in traditional board games. Researchers have turned their attention to modern video games. These games are more challenging and also more close to real-life tasks for artificial intelligence agents. In the recent years, methods based on deep reinforcement learning showed promising future. Among the works, Deep Q-Networks (DQN) stood out. DQN was able to play a lot of the Atari video games with better than human performance. And it has done this task using visual inputs just like what we human beings do. Deep reinforcement learning also displayed potential to be excel in other game environments, for instance, DoomViz, StarCraft, StarCraft II, and TORCS.

However, there still exist problems that make deep reinforcement learning difficult to be applied in more complex game environments. Although DQN showed stable learning and better than human performance in many Atari games, it was still not able to solve hierarchical tasks efficiently. The existing deep reinforcement learning methods have also been proved unstable in learning to complete tasks in continuous action space. Besides, when the actions are parametrized, agents perform poorly as well. Another problem is no efficient and general method is designed to handle multiple game agent or game units at the moment.

To solve these questions and improve the stability and performance of deep reinforcement learning algorithms. Researchers need to try out existing methods in more complex and close to reality environments. By doing this, weakness of existing algorithms can be exposed. We can then work out more efficient algorithms and more stable deep network architectures to cope with features like continuous action space and sparse reward.

In this work, we focus on a very complex and popular game environment, the StarCraft II learning environment. We proposed a new architecture to allow agents to learn to control multiple units more efficiently. Moreover, we practice transfer learning on our architecture to allow agents to learn to complete a difficult task from completing simple tasks. By doing

these, we proved that deep reinforcement learning could complete relatively complicated tasks in an environment with complex, multi-unit, and continuous action space features. With this work, we want to get one step closer to the application of deep reinforcement learning in real life tasks.

Table of contents

List of figures xi

List of tables xiii

Nomenclature xv

1 Introduction 1

1.1 Background: Games and AI Research . . . 1
1.2 Purpose of This Work . . . 2
1.3 Contributions of This Work . . . 2
1.4 Structure of This Thesis . . . 3

2 Background 5

2.1 Markov Decision Process . . . 5
2.2 Reinforcement Learning . . . 6
2.3 Artificial Deep Neural Networks . . . 8
2.3.1 Convolutional Neural Networks (CNN) . . . 9
2.3.2 Q-Learning and DQN . . . 11
2.3.3 Actor-critic method and A3C . . . 12
2.4 The StarCraft II Learning Environment . . . 14
2.4.1 State Space . . . 15
2.4.2 Action Space . . . 15
2.4.3 Minitasks . . . 16
2.4.4 Baseline Methods for SC2LE . . . 18

3 Proposed Method 19

3.1 MLSH-like Architecture . . . 19
3.2 Unit Selector . . . 22

4 Experiments and Results **29**
4.1 General Experiment Settings 29
4.2 Results 29
4.2.1 Performances and Analysis 30
4.2.2 Agent Behaviour and Analysis 30

5 Conclusions and Future Work **33**
5.1 Conclusions 33
5.2 Future Works 33
5.2.1 More Complex Mini-tasks 34
5.2.2 The Quantification of Multi-unit Cooperative Behaviours 34
5.2.3 Playing Full Game of StarCraft II 34

References **37**

Appendix A Research Environment Set-up **39**

Appendix B Details of Experiment Hyper-parameters **41**

Appendix C SC2LE Actions and Action parameters **43**

List of figures

2.1 How an RL agent interacts with the environment (StarCraft II). 7
2.2 How a DQN RL agent interact with the environment (StarCraft II). 10
2.3 How an RL agent with policy network interacts with the environment (StarCraft II). 13
2.4 Feature layers in Linux 17
2.5 Structure of the baseline fully conv network 18

3.1 Structure of mlsh-like Network 21
3.2 Training Flowchart of mlsh-like structure 21
3.3 Structure of Unit Selector Network 22
3.4 Training Flowchart of mlsh-like algorithm of unit selector model 23
3.5 Training Flowchart of solo update algorithm of unit selector model 25
3.6 Training Flowchart of combined update algorithm of unit selector model . . 26

List of tables

4.1 Average score of each agent after being trained for some episodes 30

B.1 Experiment Hyper-parameters 41

C.1 Categories of Actions and Example Actions in SC2LE 43

Nomenclature

Acronyms / Abbreviations

CNN Convolutional Neural Networks

DQN Deep Q Learning

DRL Deep Reinforcement Learning

MDP Markov Decision Process

MLSH Meta Learning Shared Hierarchy Algorithm

NN Neural Networks

RL Reinforcement Learning

SC2LE StarCraft II Learning Environment

Chapter 1

Introduction

1.1 Background: Games and AI Research

Researchers have realized platforms for testing and displaying the capabilities of artificial intelligence agents were crucial in the beginning of artificial intelligence research. Game playing contains many interesting features such as planning, responsive decision making and competitivity. Also, games provide environments to develop artificial intelligence agents that can potentially be used beyond gameplay. Therefore, games are used as one of these platforms since the 1950s. In the past seventy years, we have seen many impressive achievements in game AI. These results were mainly accomplished in traditional board games or simulations of classic board games like Chess, Shogi or Go[2].

Ultimately, the goal of research on artificial game intelligence has one principal aim. To find a way to allow an AI agent to be able to learn and complete different tasks in various environments and rules without knowing the environment in prior. By doing this, we can further use this agent to tackle complicated tasks in real life like driving or automated manufacturing. However, existing state-of-the-art AI agents are designed to either solve a single game or to play under a fixed environment. We are still very far from this final goal of game AI research. The utilization of video games as testing environments can help use to get one step closer to this goal.

From the 1970s, computer video games started to emerge and become one of the most broadly seen form of entertainment in the society. The idea of using video games to test and display artificial intelligence agent has started since then. Compared to traditional board games, video games contain typically more complex game mechanism and difficult rules. Video games also require a much faster response from the agent. Therefore, video games require quick and responsive actions from AI agent during gameplay. Also, agents need to

act logically and hierarchically to achieve a higher score in video games. These requirements make video game AI implementation more similar to complete tasks in real life.

Using reinforcement learning to train an artificial intelligence agent to play games have been used for a long time. Recent progress in deep neural networks has to lead us one step closer to our ultimate goal. In 2015, the Deep Q Network (DQN)[9] allowed an agent to play most of the Atari 2600 games with close to or better than human performance. Following this work, many different methods have been purposed towards building up a more general game AI that can be used to play various games. Also, different types of gaming environments have been implemented and published to allow researchers to test their methods more conveniently.

1.2 Purpose of This Work

In this work, we desire to take the research of video game AI one step further towards application in real world. We try to improve the performance of Deep Reinforcement Learning based AI agents on complex game environments. We decided to carry out experiments on the StarCraft II learning environment. By doing this, we want to reveal the limits of existing deep reinforcement learning algorithms. We also want to improve on the existing algorithms and neural network structures by exceeding the limits of current algorithms. We discuss the following problems of deep reinforcement learning that have been long observed in other work [6].

- Slow and Unstable learning in environments with continuous action space.
- Unable to conduct multi-agent or multi-unit control efficiently.
- Practice of transfer learning in complex game environment.

1.3 Contributions of This Work

To fulfill the above questions, we have made the following contributions:

1. Investigate the performance of existing deep reinforcement learning in the StarCraft II learning environment.
2. Purposed a new neural network structure to be applied to the StarCraft II learning environment.
3. Explore the possibility of letting the algorithm to learn to control multiple units efficiently.

1.4 Structure of This Thesis

We will first discuss reinforcement learning, deep reinforcement learning, the StarCraft II learning environment, and other necessary theoretical background information in **Chapter 2**. We will also review some related work on in deep reinforcement learning that is either used as a component of this work or have inspired this work in the same chapter. Then in **Chapter 3**, we introduce our proposed method designed to excel minitasks in the StarCraft II Learning Environment. We describe the experiment setting and results in **Chapter 4**. Last but not least, conclusions are drawn in **Chapter 5**.

Chapter 2

Background

In this chapter, we will present the necessary theoretical background of our work. This chapter should be understood as a precursor of the following chapters. We first describe the concepts and theoretical backgrounds of deep reinforcement learning. This includes what the Markov Decision Process is and how the reinforcement learning scheme is built on it. We then talk about what is a neural network and how it can be used as a function approximator in the reinforcement learning process. This part of the section of the chapter is finished with a particular form of neural networks, the convolutional neural networks. This kind of neural networks is used in many parts of both our purposed solution and baseline methods.

In the latter part of this chapter, we introduce the StarCraft II Learning environment. We first describe how a human player may play the game. Subsequently, we describe the reinforcement learning related features of this environment. In this work, we do not play the full game of StarCraft II because it is far out of the ability of all existing artificial intelligence agents.

2.1 Markov Decision Process

In this section, we present the Markov Decision Process (MDP). MDP is the mathematical framework models sequential decision-making problems.

In most of the traditional board game gameplay, for example, go or chess. Players need to make a decision in each round. Then a move is made based on the decision. To win the game, these decisions need to be highly logical. Ideally, A decision should be made on the player's judgment of all factors in a game. To use the game of Go as an example, the player may need to think about the board situation, the opponent's level of play, the possible moves the opponent may take, and even the physiological and psychological conditions of the opponent. In video games, we can still monitor the game with a similar approach. We

model the game as a sequence of discrete units called **timesteps**. At each timestep, our agent makes a decision based on the **state** of the game in that timestep. Similar to making a move in board games, the agent then perform an action to interact with the environment.

The ideal mathematical formalization of the above process is called the Markov Decision Process (MDP). In this thesis, we will use the terminologies defined in the MDP to describe our method. We now present the MDP as below:

- The learner is called the *agent*.
- Everything the agent interacts with is called the *environment*
- The agent receives *state* from the environments that indicates the situations in the game or task
- The agent then select an action and the environment reacts to the action and replies with a new state.
- The environment also gives rise to a numeric value to the agent called *reward*. The agent needs to maximize this value
- This interaction continues until the game ends

Finally, an MDP can be treated as a 5-tuple (S, A, P, R, γ). S denotes a set of states that is possible to be found in the environment. In each timestep, the agent receives current state $s \in S$. A represents a set of available actions for the agent. The agent needs to choose an action $a \in A$ at every timestep. P denotes the transition function where $s' \sim P(s, a)$

. This function shows the transition relations when the environment receives an action. R stands for the reward function, where $R : S \times A \to \mathbb{R}$. Finally, γ stands for the discount factor during reward calculation. γ is a fraction that decides if an agent wants a short-term reward or long-term reward.

2.2 Reinforcement Learning

In this section, we describe our core method, reinforcement learning. Reinforcement learning is a branch of machine learning. It is the unsupervised process of an agent learn to choose a sequence of actions to maximize cumulative reward in an environment. More formally, the agent needs to learn a policy π. π is a mapping between states $s \in S$ to actions $a \in A$. This policy aims to maximize the cumulative reward during the process. Sometimes we use the discounted cumulative rewards to adjust as the target to maximize. Discounted cumulative

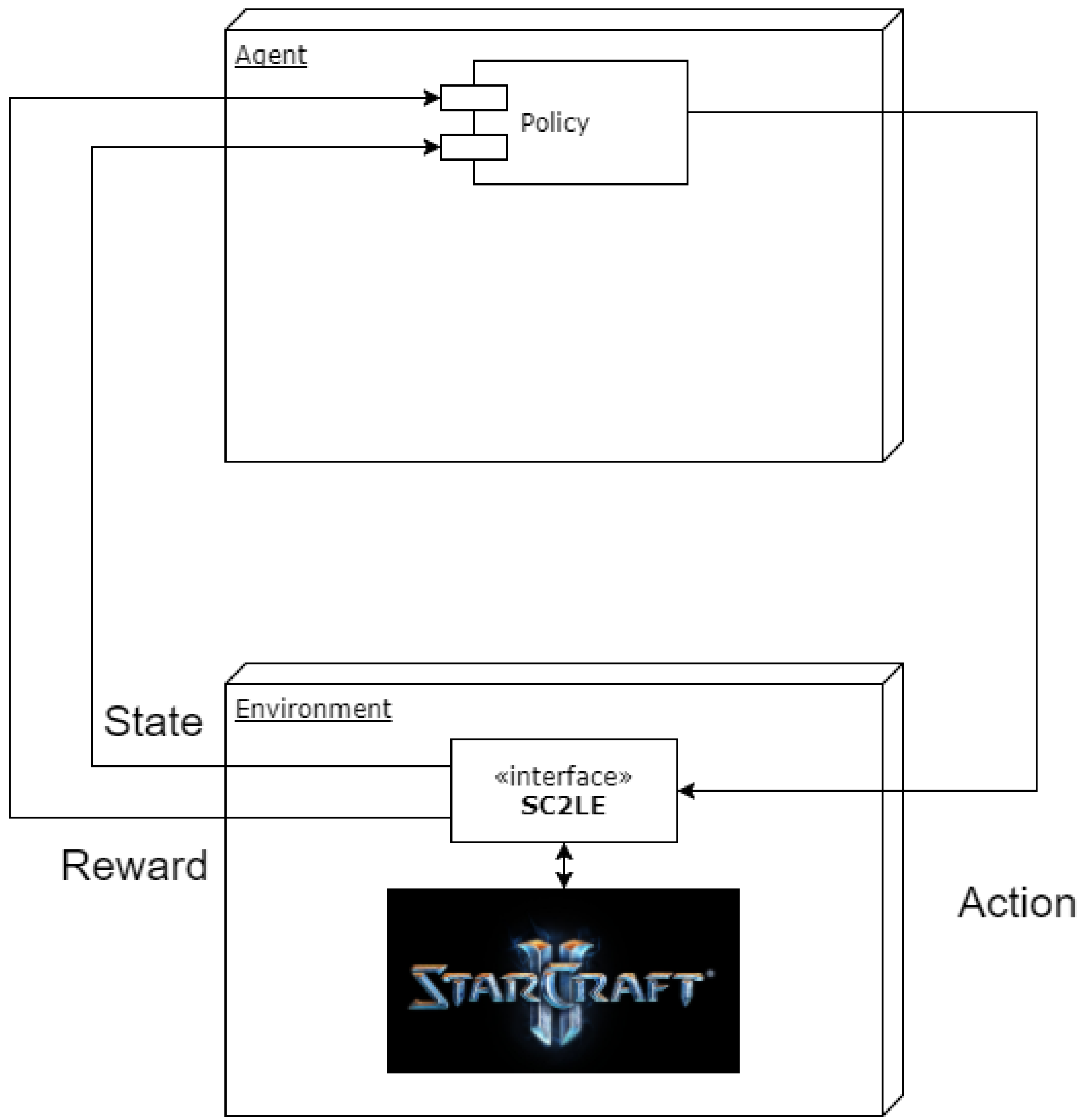

Fig. 2.1 How an RL agent interacts with the environment (StarCraft II).

reward J can be represented as $J = \sum_{t=0}^{\infty} \gamma^t r_t$, where γ is the discount factor. It is normally between 0 and one and indicates if the agent seeks long-term return or short-term return.

There are many ways to learn an optimal policy that can maximize the discounted cumulative reward. In this work, we will review only two methods, Q-learning, and actor-critic. Before we dive into these, we first introduce Artificial Deep Neural Networks. Combining deep neural networks and reinforcement learning was the key to success in the recent advances in game AI.

2.3 Artificial Deep Neural Networks

Artificial neural networks are computing structure originally inspired by the biological neural networks. However, artificial neural networks are fundamentally different from actual biological neural networks. In this part of the thesis, we briefly review the capability of this structure. Furthermore, we discuss how it is used for reinforcement learning. This section contains a brief review of DNN. We are not going to talk about detailed implementation and update scheme.

Artificial neural networks are built upon a basic computation structure called the neuron or node. Neurons normally carry out the following simple computation:

$$y = a(Wx + b) \tag{2.1}$$

x is the input value, y is the output value, W and b are numeric values named as weight and bias. In this thesis we will denote θ as the set containing all W and b in a DNN, $\theta = (W_1, b_1, W_2, b_2, ...W_k, b_k)$. a is a non-linear function, namely the activation function. Activation functions can be any non-linear function. Throughout this work we use the ReLU [8] activation function.

An array of neurons forms a layer in the neural networks. These layers are then stacked and connected to form the whole neural network. In short, a deep neural network computes the following:

$$\hat{y} = a_k(W_k...a_1(W_1x + b_1)... + b_k) \tag{2.2}$$

Where x, W, and b are normally matrix containing the inputs, weights, and biases of each layer. $haty$ is the prediction output of the neural network. The network can be trained by comparing $\hat{y}$ with the correct label y. The specific algorithm to carry out DNN training is called the Backpropagation algorithm (Werbos, 1974)[11].

Backpropagation algorithm minimizes a function defined by the implementer. This function to be minimized is called the Loss function, L. Typically, L tends to minimize the difference between the predicted value, $\hat{y}$ and the actual value, y. Different ways to calculate difference can be used. For instance square error or cross entropy. One of the simple forms of L can be presented as:

$$L = |\hat{y} - y| \tag{2.3}$$

Backpropagation provides a way to calculate gradients $\Delta_\theta(L)$, to update W and b in neural network so L could be minimized. When applied to the parameters, these gradients were multiplied by a fraction called the learning rate. Learning rate is normally an adaptive value. we use the Adam optimizer [3] in our work to adjust the learning rate.

Different ways of structuring and connecting different layers in DNN exists to serve different purposes to use the neural networks. But generally, fully connected ones can normally be found in most of the deep neural networks implementations. In later sections, we are also going to talk about Convolutional Neural Networks (CNN). Which is used widely in video game playing as well as imaging recognition. Last but not least, one of the most crucial fact about DNN we should note is we do not understand the mechanism of how DNN generates the predictions. The mathematical background is lacking so during our training and tuning of a DNN we usually are guessing.

2.3.1 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) (LeCun et al., 1998) is a Deep Neural Network structure designed to fetch key information from images. CNN can learn regional or local information from an image. It achieves this by learning low-dimensional filters. CNN has first been proved successful in doing static image recognition tasks. It was then used in many successful applications of deep reinforcement learning. These include some methods we are going to use in this work. Namely DQN[9], A3C, and baseline methods from the StarCraft II Learning Environment. These methods will be introduced in the following sections in this chapter. As you will see later, our major testing environment StarCraft II Learning Environment uses images to represent a state of the game. Therefore, we need a mechanism to "see" these images. CNN has been proved to serve this purpose perfectly in previous works or Deep RL. Hence, we use CNN layers as the input layers of our DNN model.

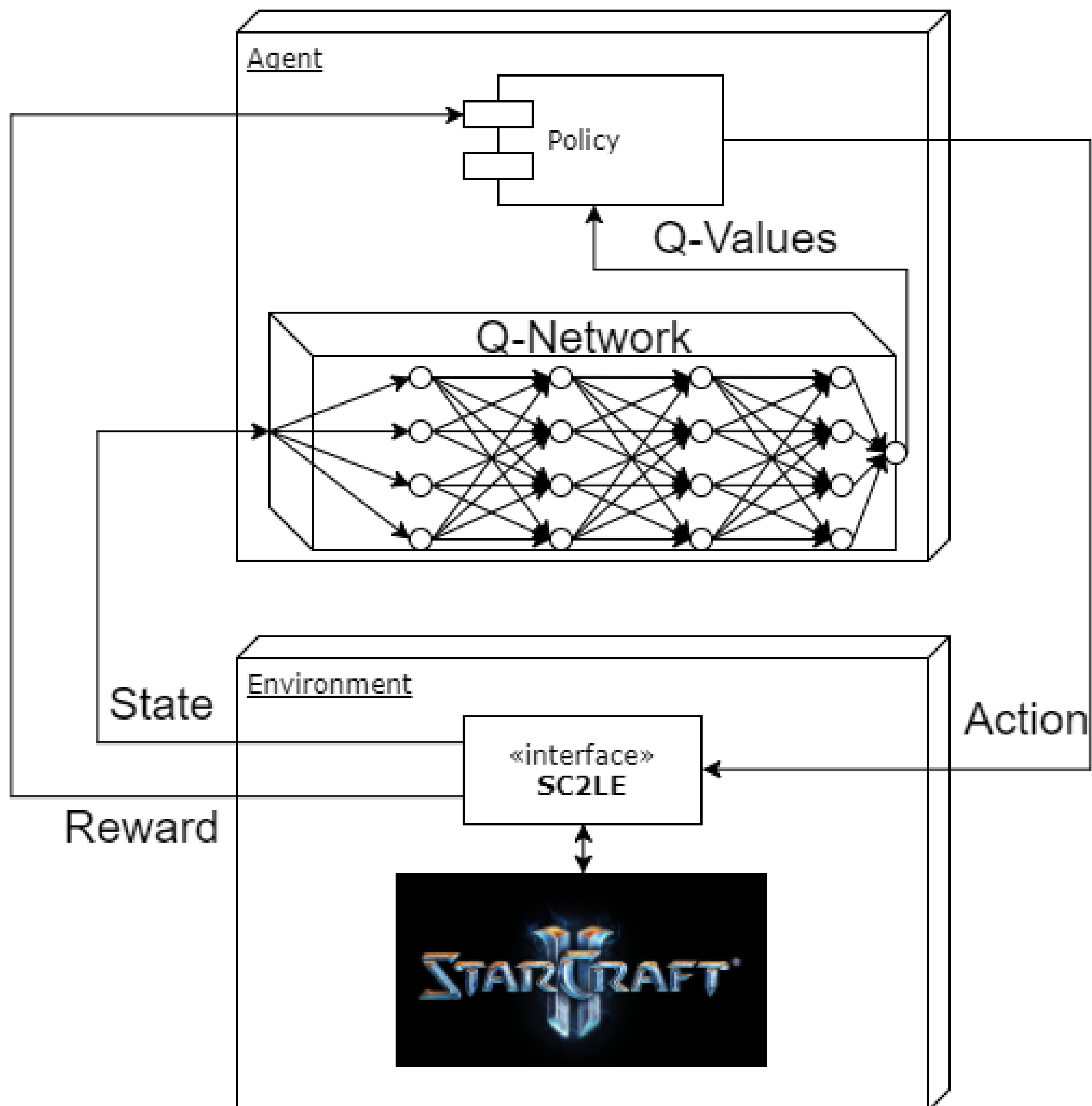

Fig. 2.2 How a DQN RL agent interact with the environment (StarCraft II).

2.3.2 Q-Learning and DQN

To find an optimal policy, we let our agent learn a value to estimate how much discounted cumulative reward each action can potentially bring to the agent under different states of the game. This value is called *Q value*, $Q^{\pi}(s,a)$. After knowing each action's Q value, we can select the action with the greatest Q value to ensure we maximize the discounted cumulative reward.During the learning process, Q values are iteratively refined by playing the game and visiting the same s,a pair in different episodes of the games. An intuitive way to update Q values from immediate reward can be described by the Bellman Equation (Bellman, 1957):

$$Q(s,a) = R(s,a) + \gamma Q(s',a') \tag{2.4}$$

The algorithm using the Bellman Equation to update Q values is called SARSA (Sutton and Barto, 1998)[7]. By repeatedly playing the game and updating all Q values iteratively, the Q value estimates are likely to converge into a very close estimation, Q^*. Hence, selecting actions using Q^* gives an optimal policy.

In our work we use a technique called Q-Learning (Watkins and Dayan, 1992)[1], which updates Q values with a modified version of the Bellman Equation:

$$Q(s,a) = R(s,a) + \gamma max_{a'} Q(s',a') \tag{2.5}$$

The Q-Learning scheme uses the maximum Q value in the next state to carry out an update. The trick here is that the action with $max_{a'}Q(s',a')$ may not be selected in the next state by the agent. This update scheme allows the agent to explore potentially bad moves. The agent may, therefore, find more possible moves. Also due to the possibility of selecting a Q value containing an action that may not be taken, this way of updating is called off-policy update.

How Q-Learning is combined with deep neural networks to estimate the Q value is described in the next section.

Deep Q Network

Deep Q Network (DQN, Mnih el al.,2015) is the first pieces of work that combines RL and deep neural networks to learn game playing in discrete action spaces. It extends Q-Learning we described above with a deep neural network as a Q-function approximator. DQN was implemented to play games in the Atari 2600 environment as gained great success.

For a bit of recap, Q values are state-action pairs. Simpler games a finite amount of game states. So there would be a finite amount of state-action pairs. Therefore, we can store the Q values in a table or a database. And use them as a function to get Q-values. However, video

games have an almost infinite amount of states. DQN inspirationally used a deep neural network to approximate the relation between state-action pairs and Q values. This DNN takes the state-action pair as input and outputs the Q value for each action. The neural network is parametrized by a set of weights and biases, denoted by θ. Hence, Q values estimated by this DNN can be denoted as $Q(s,a|\theta)$. Once θ is trained, we will have reliable estimator of Q function $Q(s,a|\theta)$. And our agent can issue actions according to the Q-learning schema. The Loss function of DQN can be presented as:

$$L(s,a|\theta_i) = (r + \gamma max_{a'} Q(s',a'|\theta_i) - Q(s,a|\theta_i))^2 \tag{2.6}$$

$$\theta_{i+1} = \theta i + \alpha \Delta_\theta L(\theta_i) \tag{2.7}$$

Where γ is the discount rate, and α is the learning rate. They are both a fraction.

The actual update, is carried out by repeatedly playing the game and gather a tuple called the experience, $e_t = (s_t, a_t, r_t, s_{t+1})$. A few critical techniques are used to solve different issues affects learning stability:

- The game experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ is collected and stored in a buffer. Updates on these experiences are processed by randomly sampling this buffer. This avoids the agent to be biased too much in some extreme situations.
- During the early stages of the updates, the agent sometimes performs random actions. This schema allows the agent to explore more state spaces in the game environment.

2.3.3 Actor-critic method and A3C

sNext, we look at another important algorithm of RL in our work, the actor-critic architecture (Sutton and Barto, 1998)[7]. This algorithm solves the problem that DQN can only be used to select an action in discrete action spaces. In continuous action spaces, there is an infinite amount of actions. Therefore, our DQN, which outputs Q value for each action is no longer tractable.

Actor-critic architecture normally consists of 2 Neural networks. The actor network, or sometimes called the policy network $\pi(a_t|s_t;\theta)$. This network outputs an action based on the input state. The second part is the critic, or sometimes called the value network $V(s_t;\theta_v)$. This network takes a state as input and outputs a numerical evaluation of how good the state

is. The value function is used to replace $max_{a'}Q(s'a'|\theta_i)$ part of the Q learning loss function:

$$L(s,a|\theta_i) = (r + \gamma max_{a'}Q(s',a'|\theta_i) - Q(s,a|\theta_i))^2 \tag{2.8}$$

And the Loss function for actor-critic architecture is:

$$L(s,a|\theta_i) = (r + \gamma Q(s',V(s'|\theta_v)'|\theta_Q) - Q(s,a|\theta_Q))^2 \tag{2.9}$$

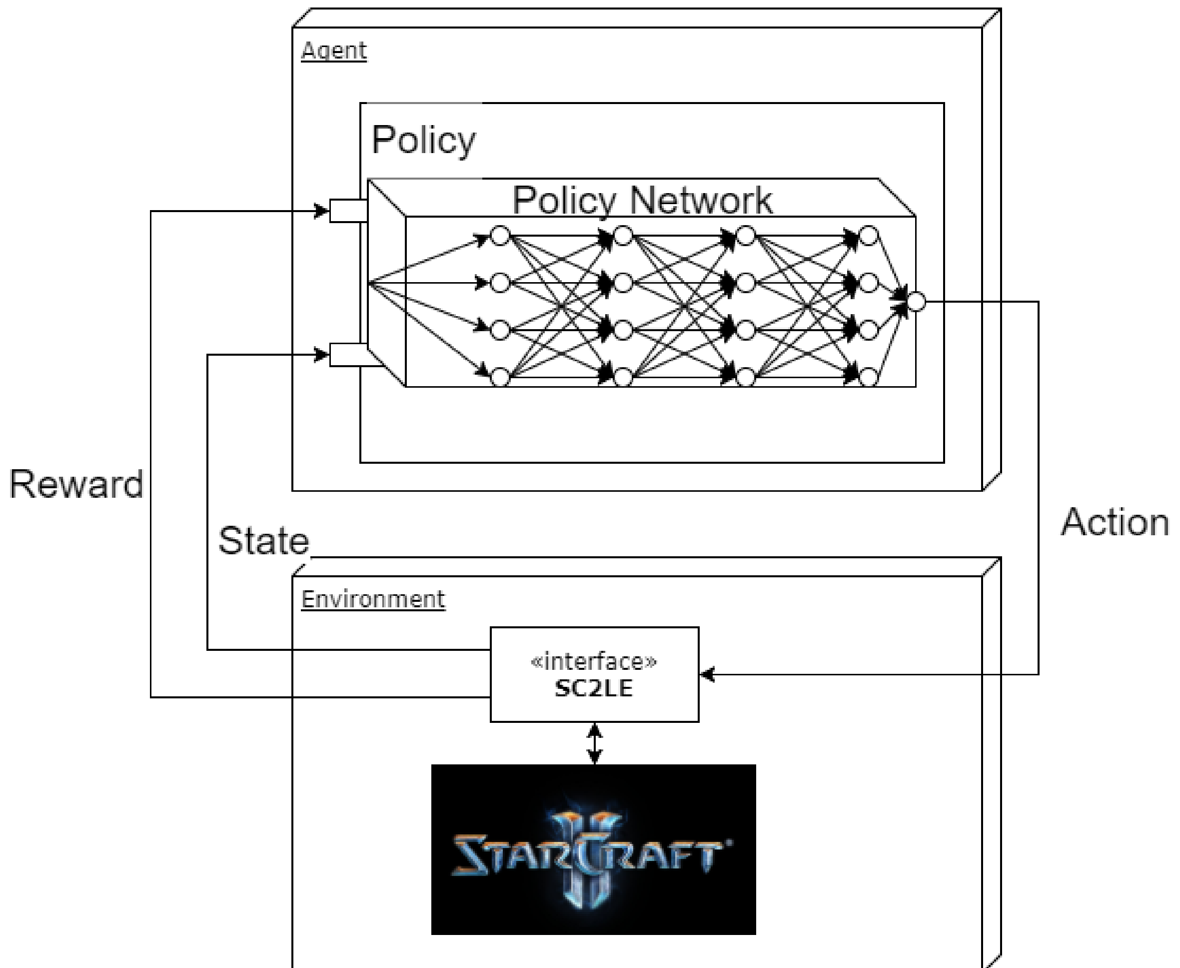

Fig. 2.3 How an RL agent with policy network interacts with the environment (StarCraft II).

Policy network will use the knowledge of the value network to calculate loss. In our work, our baseline method is based on a variant of actor-critic, the Asynchronous advantage actor-critic algorithm (A3C, Mnih et al., 2016)[10]. This algorithm collects experiences

using multiple threads and carry out update to the methods asynchronously. Details of this algorithm will be introduced in the baseline methods section later in this chapter.

2.4 The StarCraft II Learning Environment

We now introduce the StarCraft II Learning Environment (SC2LE)[4]. This is the major testing environment for our research.

StarCraft II is a real-time strategy game released by Blizzard Entertainment in 2010. It is considered one of the most popular game in the history of video games. Besides, StarCraft II is also considered as an electronic sport and have been played competitively all around the world. From 2010 to early 2018, competitive StarCraft II tournaments and leagues have a cumulative prize pool of $ 25,163,366.06 (esportsearnings.com, 2018). This comes to the fourth largest cumulative prize pool in all competitive video games in the world. StarCraft II Learning Environment (SC2LE) is an API interface built on the game to allow computer agents to interact with the game. SC2LE was released in Aug. 2017. SC2LE is designed to carry out There exist little research on the environment so far since the environment has been released for only a few months. However, due to StarCraft II's competitive nature, the game holds good features to be a testing field for Artificial Intelligence agents. These features are rarely seen or different from other existing Deep RL testing environment(e.g. OpenAI gym, MuJoCo).

- Complex state space and action spaces. Detailed state and action spaces setting will be mentioned in the later sections.

- The agent needs to control multiple units in StarCraft II. So far all the successful game AI agents are made to control a single unit.

- Requirements of fast response. Top human players may issue over 200 actions per minute in the game. We would imagine an AI agent needs to has similar response speed.

- Strategic planning is required to win the game. An agent needs to think about a lot of game-related factors such as resource allocation, unit production, and unit controlling at the same time.

- StarCraft II is an imperfect information game. Most of the in-game information is covered by a mechanism called 'fog of war.'

- StarCraft II is considered one of the hardest video game to play for human players. Just like the game of Go is considered as the hardest board game.

The API is open sourced (https://github.com/Blizzard/s2client-proto). In this work we use the python wrapper (https://github.com/deepmind/pysc2) to carry out implementation. In the next few sections we will introduce how SC2LE deal with RL related features.

2.4.1 State Space

The SC2LE represent the current game state majorly with an abstracted game image input called 'feature layers'. Feature layers are a set of images with $N \times M$ pixels. Each feature layer contains information about some specific game feature. For instance: unit type, unit hit point, unit owner,and visibility. The original RGB features of StarCraft II was removed from these images. Instead the colours in the feature layers contains information such as hit point amount. The use of feature layers remove useless graphical peripherals for the AI agent but keeps the spatial and graphical information.

Feature layers are further divided into two categories, namely the *screen*, and the *minimap*. The screen is a detailed view of the current game screen. Similar to what a human player sees from the game, screen feature layers only contains a subsection of the whole game map. Agent can execute most of the available actions on the screen. The minimap feature layer contains information correspond to human player's minimap. It is a coarse view of the whole game map. Some game feature information are held by both of these two categories of feature layers. But there also exist some game features are unique to only one category.

Other than the feature layers, SC2LE also provides the agent some non-spatial information. In human interface these information are also listed around the main screen of the game. Examples of these information includes the current amount of gas and minerals (in-game resources) the player has and the set of available actions for the currently selected unit.

2.4.2 Action Space

Actions in SC2LE are designed to mimic what human can do with StarCraft II game interface. In total, 523 different actions can be issued by the agent. These actions can be put into two categories:

- Spatial Actions: These actions mimic what human players may do on the main screen with the mouse. Therefore, not only the action name, but also some action parameters. The actual type and number of actions varies greatly depends on the action type. Typically the arguments will include one or two coordinates (x, y). These coordinates

indicate the point on the screen where action is executed. A list of action types and their related parameters can be found in Appendix C.

- Non-spatial Actions: These action are normally related to special in-game unit skills like morphing. Or some specific actions like stop current action and. These actions require no coordinate related parameters. But sometimes require a boolean or scalar parameters.

One important thing to notice is how we define timestep in the game. By theory we can use one game frame as one timestep. In StarCraft II, 30 frames are played every second. Therefore, following this definition the agent will be able to carry out 1800 actions per minute. Whereas human players normally issue 30 to 300 actions per minute. This makes the agent having an unfair advantage against human players. Therefore, the official suggested timestep is 8 game frames. This allows the agent to issue around 180 actions per minute.

2.4.3 Minitasks

Playing the full game of StarCraft II requires a lot of skills like micro-management of units, planning, and game mechanism understanding. Therefore, no learning based artificial intelligence agent is feasible of plying a full game of StarCraft II effectively. Vinyals et al.[4] have tested some of the baseline methods to play the full game, and all of them performed poorly. Hence, at this point, we believe there is no point in trying to solve a full game of StarCraft II with deep reinforcement learning agents.

Alongside the ability to allow AI agents to play the full game of StarCraft II, SC2LE also provides a set of standardized mini-games to allow agents to learn to play simpler tasks in the environment. These tasks have pre-set reward scheme and can be run repeatedly. They also take a short time to complete, normally within 3000 in-game timesteps. Although the actual time required depends on the computation power of your hardware, It normally takes less than 5 seconds per episode. Unlike the partially observable full games of StarCraft II, mini-tasks are fully observable. Hence, the difficulty for agents is further reduced.

In this work, we focus on solving the 'Collect Mineral Shards' task for our agent and uses the 'Move to Beacon' task as an auxiliary task to help our training of agents.

Move To Beacon Task

This is the most simple task in the officially provided mini-tasks in SC2LE. It requires the agent to control one unit to move towards a spot on a small size map. After reaching the spot, another target spot will be generated randomly on the map again. The agent needs to move

to the new spot. The agent receives a reward for each spot it visits. The game ends when a pre-set amount of in-game timesteps are reached.

The task is the very first task every deep RL agent should run for SC2LE. And if implemented correctly, most of the state-of-the-art RL agents can complete it with close to human performance.

Collect Mineral Shards Task

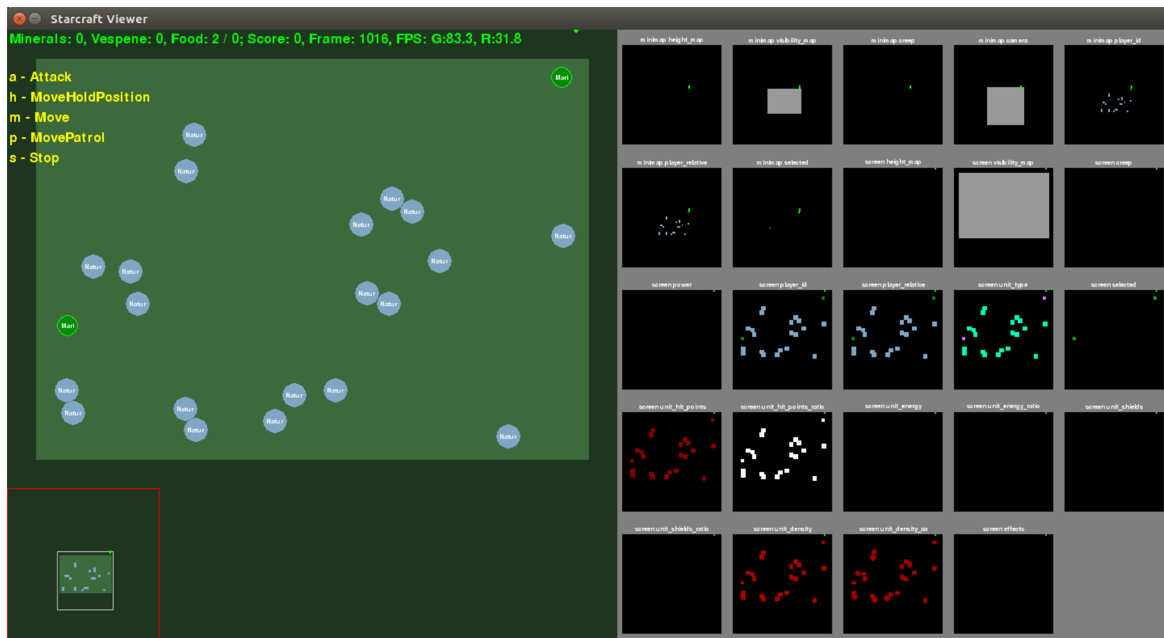

Fig. 2.4 Feature layers in Linux when Agent is playing the Collect Mineral Shards task

In the Collect Mineral Shards task, the agent controls two units. These units will be generated on the map on two random spots. Besides the two units, twenty 'mineral shards' will also be generated randomly on the map. The agent needs to control the two units to move to the mineral shards. When a unit gets close enough to a shard, the unit will collect the shard. Once collected, the shard would disappear from the game. Each collected shard provides one reward for the agent. After all twenty shards are collected, another 20 shards will be generated randomly on the map. The task continues until the pre-set timestep limit is reached.

This task is very well designed. It covers a lot of unsolved problems for deep RL.

- Multi-unit or multi-agent decision making is involved. Most state-of-the-art methods in deep reinforcement learning are designed to control a single agent or a single unit. Although in this task the agent needs to control only two units. It still expands the

action space for the agent greatly. To get to an optimal policy. The agent needs to control both units effectively without increasing the computation power dramatically.

- To collect mineral shards effectively the agent is essentially learning to solve the traveling salesman problem. This problem is a well-known NP-hard problem. Also, this problem may be more difficult to find optimal solution with two 'salesmen'. sf

2.4.4 Baseline Methods for SC2LE

Deepmind provided a few baseline methods based on A3C With the publishing of SC2LE. Just as their name states, these methods are served as baselines to be compared with for future research. In this work we implemented the FullyConv baseline. In this section we will present this method. The Neural network architecture can be found in Fig.2.3. This network

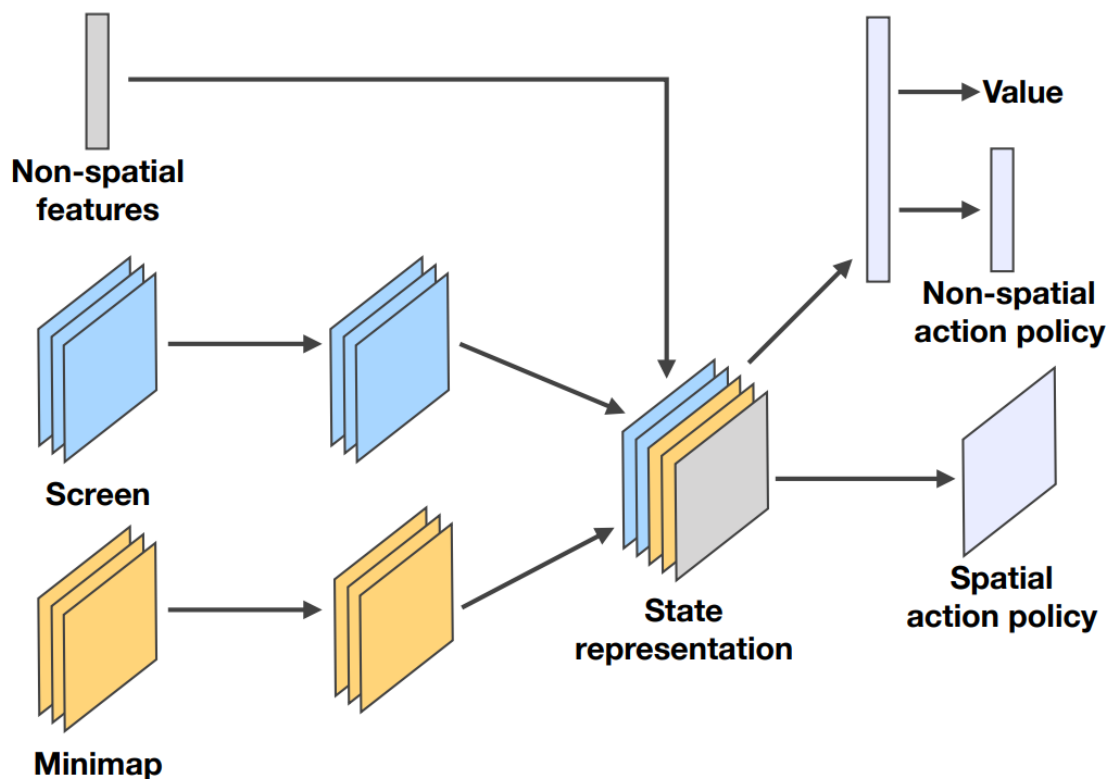

Fig. 2.5 Structure of the baseline fully conv network[4]

is updated with a A3C algorithm. Unlike normal A3C implementation, the value and policy networks in the base line method is the same network with different output layers. The input layers has 3 parts. Two CNNs reading the screen and minimap graphical inputs, and one fully connected network reading the other information array input.

Chapter 3

Proposed Method

In this Chapter we will look at three different methods, we implemented in search of improving our agent. The focus of this work is to improve the multi-unit performance and efficiency in the Collect Mineral Shards task, which is described in Chapter 2.

We first implement the SC2LE baseline method (Vinyals et al., 2017)[4]. We desire to run the baselines under our hardware and hyper-parameter settings. This is because Deep RL in continuous action and state space have been proved very unstable and be affected by many factors (Hendersen, 2017)[6]. Therefore, implementing them can ensure their performances as baselines are more accurate.

In this work, we focus on improving the multi-unit efficiency during gameplay. This is because we observed that the baseline methods could not utilize the two units given in the task efficiently. A human player would easily figure out how to use each unit to collect different minerals on the map. However, The two agents move together during game-play. The reason for this behaviour will be analysed in Chapter 4.

We attempt to improve this behaviour by using an MLSH-like architecture for our model. We imagine each policy we trained will be able to command one unit to collect minerals. And the master policy will be able to coordinate the two subpolicies to work together. We are worried about if training MLSH-like structure in the Collect Mineral Shards We also attempt to pre-train subpolicies in the Move to Beacon task. So they will have some prior knowledge about the SC2LE environment. However, the behaviour of the agents has not changed.

We analysed the reason for the agent behaviour and purposed a

3.1 MLSH-like Architecture

The Meta Learning Shared Hierarchies algorithm (MLSH, K. Frans, et al., 2017)[5] proposes a hierarchical architecture between different algorithm. MLSH algorithm consists of one

master policy and several subpolicies. In our application, we adapted their implementation to use DQN as the master policy and the baseline FullyConv A3C networks as the subpolicies. Both of the master policy module and the subpolicies take the screen, minimaps, and info array as input. Each subpolicy outputs an action based on their parameters θ. The master policy outputs a discrete array of Q values which has the length equal to the number of subpolicies. Each Q value in the Q value array estimates the return from using a specific subpolicy to generate an action. We then used the subpolicy with the maximum Q value to generate an action. We use multiple threads to collect experiences and store them in a replay buffer. Training of our networks follows this algorithm:

```
initialize all subpolicies;
initialization of all agent threads;
while Policy not converged do
    Reset master policy;
    while v is smaller than V do
        Sample Environment with different threads and store experiences in replay
          buffers;
        Update master policy using experience sampled from buffers;
    end
    while w is smaller than W do
        Sample Environment with different threads and store experiences in replay
          buffers;
        Update master policy using experience sampled from buffers;
        Update subpolicy using experience sampled from buffers if the subpolicy is
          used in this experience;
    end
end
```

Algorithm 1: MLSH Adaptation to SC2LE

Where W and V are hyper-parameters. The rationale behind using this algorithm is we expect each subpolicy will be able to control one unit in the game. The master policy will act as a coordinator between these two. Hence we can achieve multi-unit control. Following this rationale, we use two subpolicies in our architecture. We practice transfer learning using this architecture. We pre-train the subpolicies in Move To Beacon task. Using these pre-trained algorithms, we seek to increase the performance of the whole agent. We also hope by training each subpolicy alone. Each subpolicy could develop a good policy for controlling a single unit.

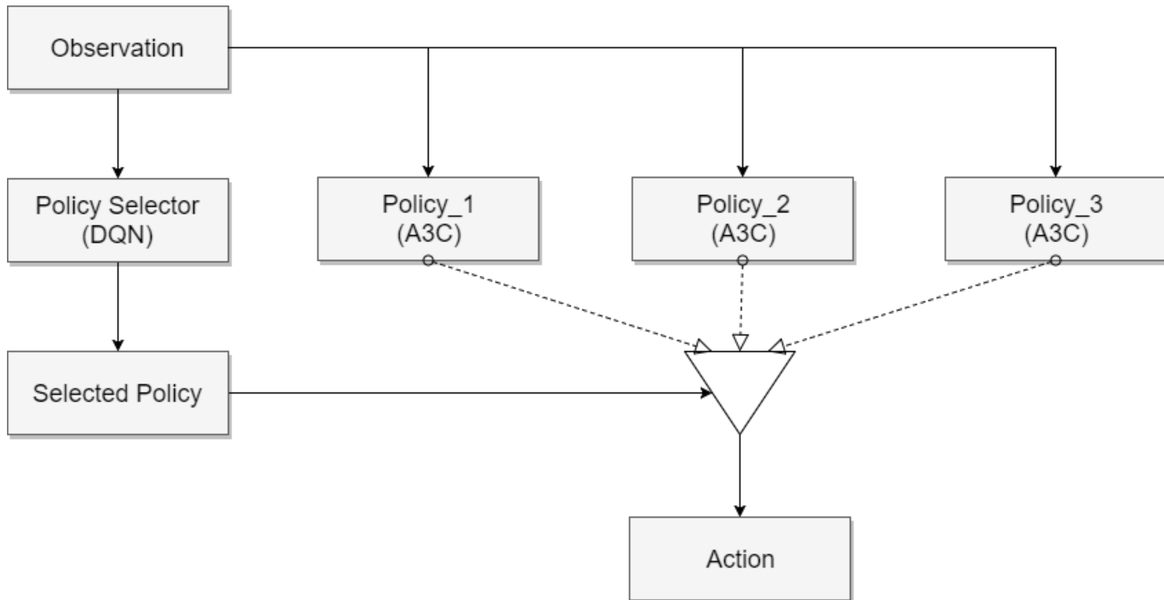

Fig. 3.1 Structure of mlsh-like Network[4]

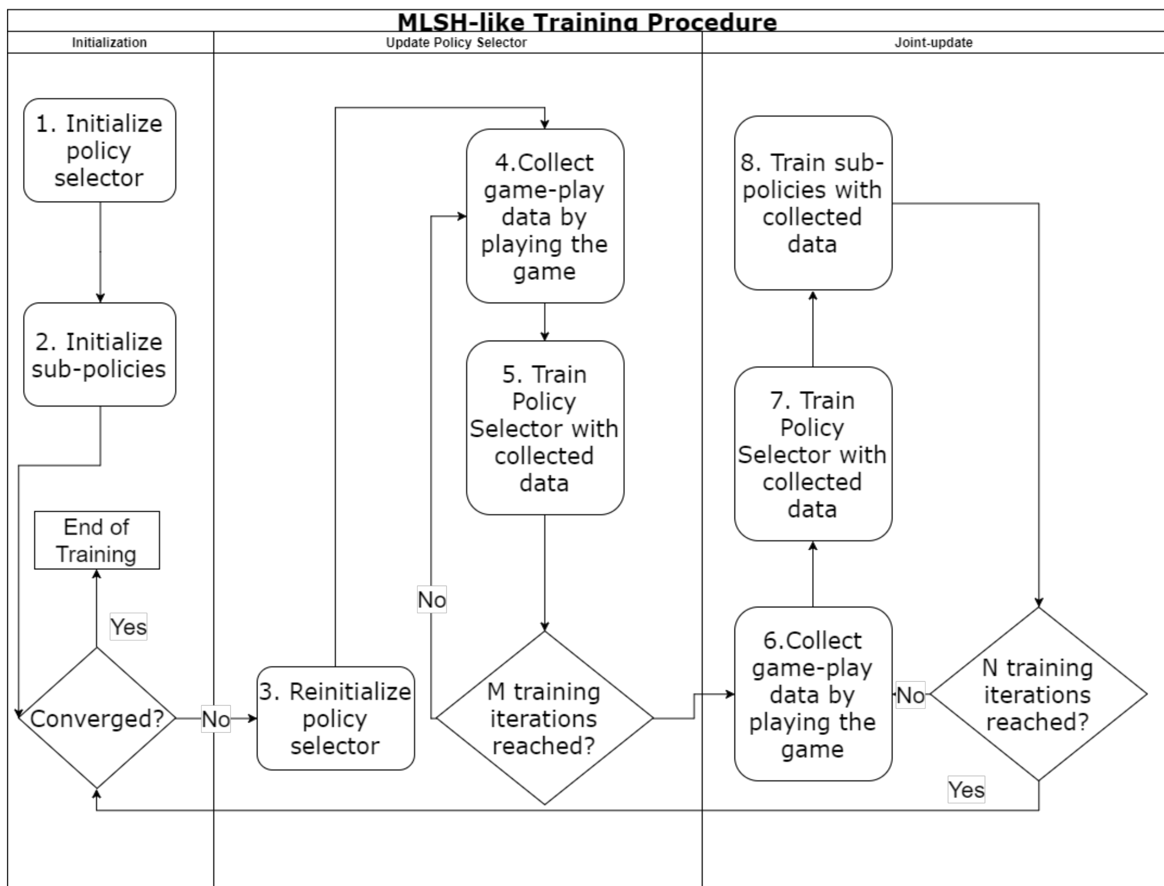

Fig. 3.2 Training Flowchart of mlsh-like structure

3.2 Unit Selector

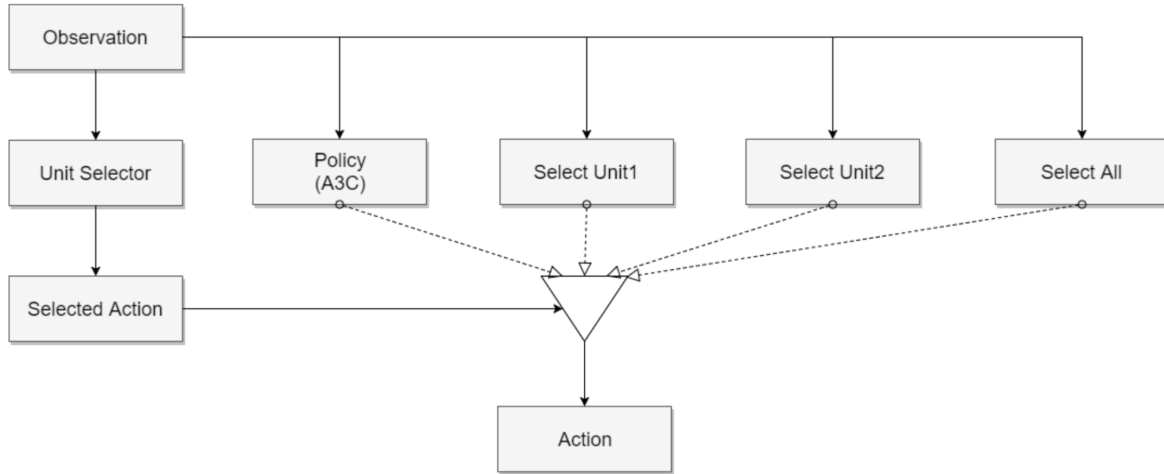

Fig. 3.3 Structure of Unit Selector Network[4]

After implementing and experimenting with the MLSH-like architecture, we have found that it increases the training speed of our agent. However, the trained agent still cannot use each unit efficiently. The behaviour of an agent is similar to what we got in the baseline methods. We therefore further purposed a unit selector architecture to seek higher multi-unit efficiency.

The unit selector algorithm consists of two modules. A unit selector module, and a policy module.

In this work, we continue to use the baseline FullyConv network as the policy module because it has been proved learning with stability in the previous experiments.

We used a DQN for the unit selector. This DQN takes the screen, minimap, and info array as input. It generates a Q value array estimating the return with the following four actions:

- Selecting both units
- Select unit one
- Select unit two
- Use policy module to generate action

This structure is expected to have the following advantages:

- The architecture has small network size. The size is only roughly two times comparing to the baseline methods.

- Potentially fast training speed since policy selector is composed with DQN and we can train it with multiple threads.
- The architecture is easy to scale. If we need to increase the number of units under control, we just need to expand the Q value array output of the unit selector.
- Transfer learning and pre-training can be applied to the policy module. So in theory, if we have good policy module for controlling the current unit under control, we can apply it directly to the agent. And the training can be reduced to train the unit selector alone.4

To train this unit selector structure, we have proposed three different training schemes. Firstly we use an MLSH-like update scheme. Unlike MLSH, we do not reset the unit selector in each update round.

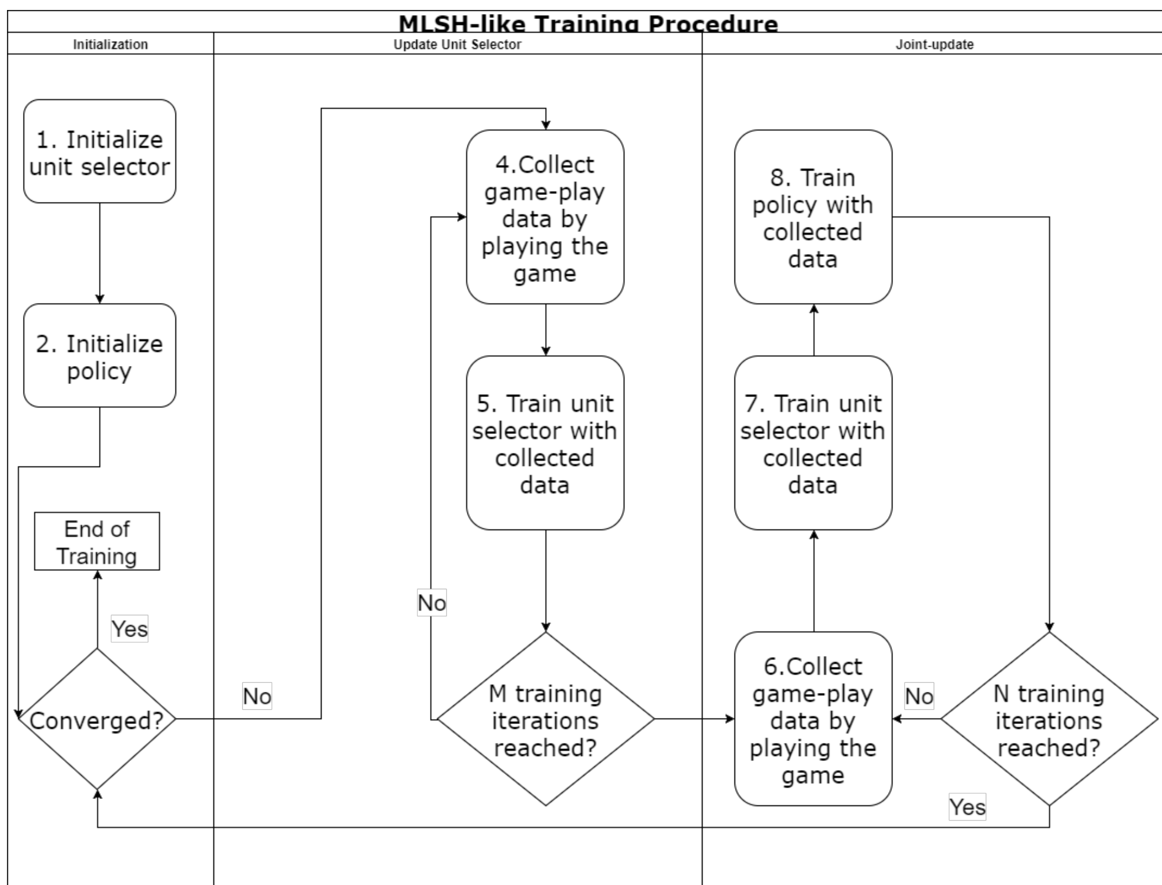

Fig. 3.4 Training Flowchart of mlsh-like algorithm of unit selector model

initialize or load policy module;
initialize unit selector module;
initialization of all agent threads;
while *Policy not converged* **do**
  while *v is smaller than V* **do**
    sample Environment with different threads and store experiences in replay buffers;
    update policy selector using experience sampled from buffers;
  end
  while *w is smaller than W* **do**
    sample environment with different threads and store experiences in replay buffers;
    update policy selector using experience sampled from buffers;
    update policy module using experience sampled from buffers if the policy module is used in this experience;
  end
end

Algorithm 2: MLSH-like Update for Unit Selector Architecture

Then we have an algorithm to only update the unit selector module with a pre-trained policy module. This training scheme is based on the assumption that if the policy module is performing well enough. We only need to train the unit selector alone.

load policy module;
initialize unit selector module;
initialization of all agent threads;
while *Policy not converged* **do**
  sample Environment with different threads and store experiences in replay buffers;
  update policy selector using experience sampled from buffers;
end

Algorithm 3: Solo Update of Unit Selector Architecture

Finally, we have an algorithm to update both modules together. The rationale behind this is even if we have a good policy module, it still needs to be adapted to the unit selector together to get a good policy.

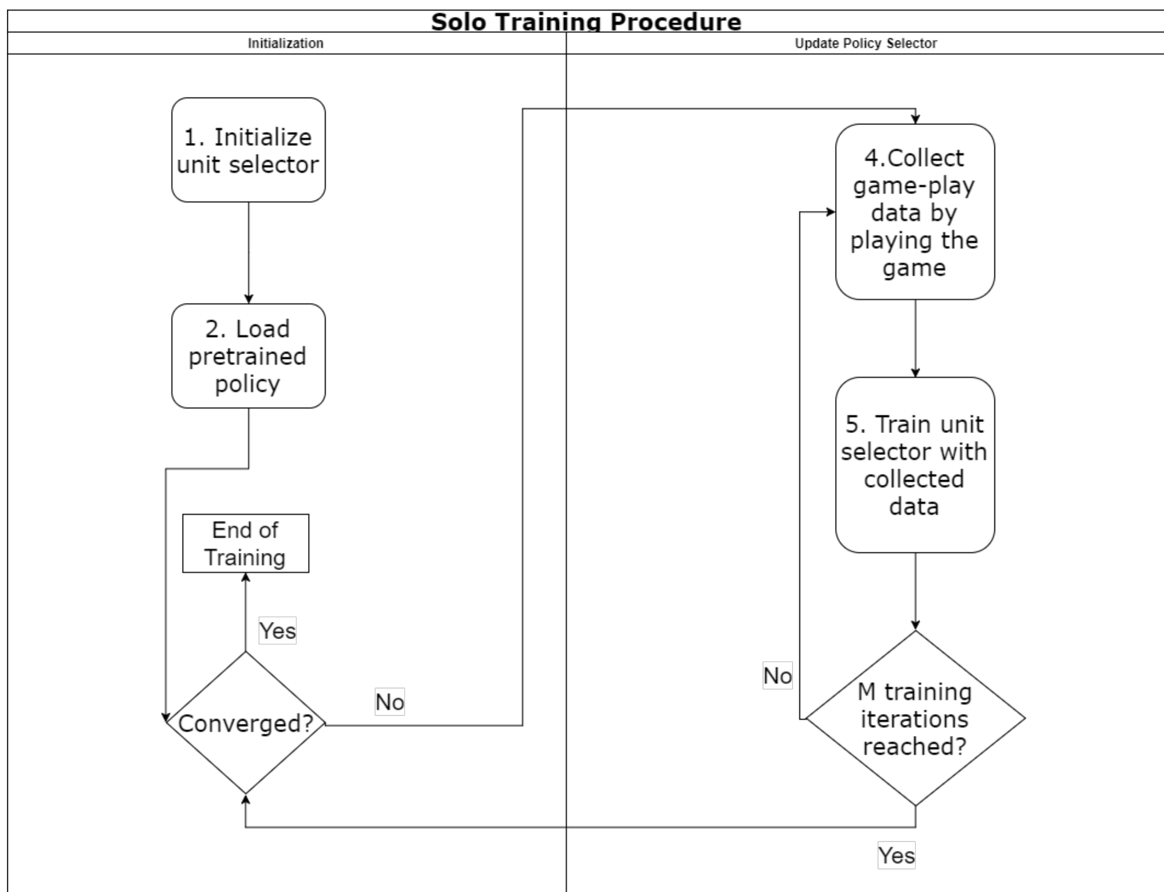

Fig. 3.5 Training Flowchart of solo update algorithm of unit selector model

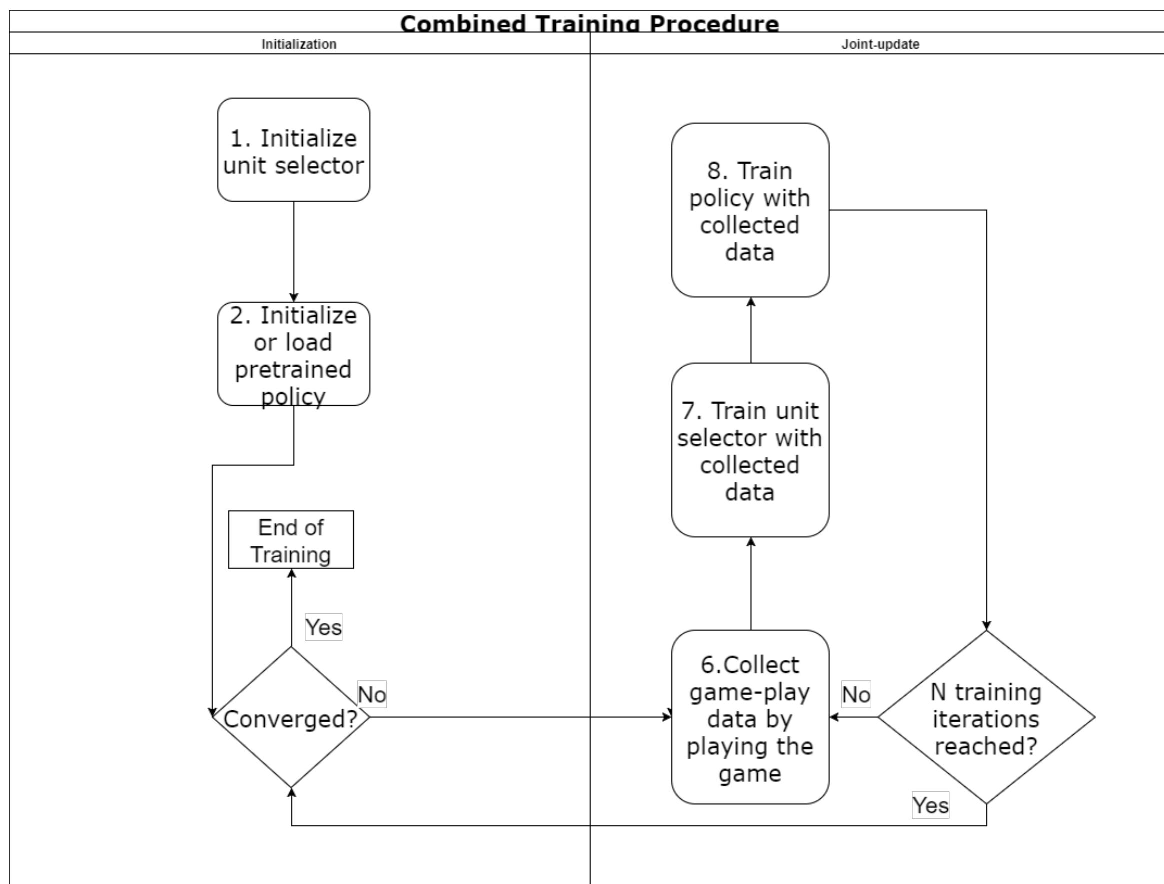

Fig. 3.6 Training Flowchart of combined update algorithm of unit selector model

```
initialize or load policy module;
initialize unit selector module;
initialization of all agent threads;
while Policy not converged do
    sample environment with different threads and store experiences in replay buffers;
    update policy selector using experience sampled from buffers;
    update policy module using experience sampled from buffers if the policy module
      is used in this experience;
end
```

Algorithm 4: Combined Update of Unit Selector Architecture

We will state our experiment settings, experiment results, agent behaviour observations, and analyses in the next Chapter.

Chapter 4

Experiments and Results

4.1 General Experiment Settings

We focus on using the StarCraft II learning environment's mini-tasks to train and evaluate our agent. The agent is trained with the purposed methods described in Chapter 4 as well as the baseline algorithm proposed by DeepMind. In order to keep baseline method's experiment result consistent with our hardware conditions, we have also ran the baseline methods on the same hardware we use to run the purposed method. All the agents in our experiment is trained with an Nvidia GTX 1080 graphics card. Due to the restriction of the hardware, for the baseline agents we can use more threads in training. For the unit selector agents, we could also use relatively large amount of threads to train. But for since the MLSH-like agents have large network size (three times the baseline when controlling two agents). Detailed experiment hyper-parameters can be found in Appendix B. The source code can be found on GitHub: https://github.com/yx3110/pysc2-agents_with_mlsh

4.2 Results

For each agent we have randomly sampled five instances from all training examples. These instances are not having better performances than others. We show the average score of the five sampled agent after they have been trained for different amount of game episodes. The final score is first taken from the mean score of playing 100 games for each instance. Then further averaged between the five instances of each kind. We have run unit selector with all three training algorithms. Due to time limitation, we can only train with the combined algorithm with pre-train data. The pre-trained data

	Agent Name	Number of Threads	Score_0	Score_1000	Score_2000	Score_3000	Score_4000	Score_5000	Score_7000	Score_10000
1	Baseline	4	1.87	2.33	5.17	4.24	9.44	12.25	20.68	32.54
2	MLSH_like	4	3.28	20.32	31.55	39.61	42.65	50.17	54.14	**65.17**
3	Unit_Sel_mlsh	8	2.15	3.41	1.23	5.32	6.77	5.87	3.23	4.23
4	Baseline	12	1.34	12.58	30.33	47.21	45.89	48.36	57.32	59.38
5	Unit_Sel_solo_pretrianed	8	12.44	25.68	24.69	32.77	33.62	32.47	30.45	39.66
6	Unit_Sel_combined	8	0.59	0.98	1.58	3.42	3.01	3.75	4.56	4.12
7	Unit_Sel_combined_pretrain	8	12.30	35.35	33.48	37.86	35.47	42.47	54.76	57.34
8	MLSH_like_pretrained	4	52.73	53.02	54.32	55.38	60.39	59.28	63.85	62.36

Table 4.1 Average score of each agent after being trained for some episodes

4.2.1 Performances and Analysis

In this section, we discuss the observation about learning speed, convergence performance about the agents during experiments.

- The performance of our baseline agent is worse than what SC2LE white paper states.

  This is a reasonable result as we have less computation power to support enough threads to run A3C more effectively.

- MLSH-like agent appears to have the best convergence performance and the fast learning speed.

  This may be due to multiple subpolicies gives it better adaptation to different states.

- The only successful way of training Unit selectors fast enough is the combined algorithm with pre-trained model.

- Although Unit_Sel_combined_pretrain have developed interesting behaviours. It does not achieve better performance in scores compared to other models.

  This may be due to the lack of coordination between the policy network and unit selector.

4.2.2 Agent Behaviour and Analysis

Despite the performance, we also focus on the behaviour about if the agent can use two different units efficiently. We found that both the MLSH-like and baseline methods appear to have the same strategy, that is to use the 2 units together. The unit selector can not totally use the two units separately. However, it has developed some interesting behaviours that increase the efficiency of using two units.

- Leaving one unit at the edge of the map. So when the new batch of minerals shards are generated. There is always one unit that can move to a closer shard.

- When the two units move together. The agent will keep the two units in a certain distance and alignment. This alignment seems to help the units collect minerals shards faster.
- The agent will sometimes let one unit to patrol around some point while it is controlling the other unit.
- One important and sad fact in this work is we don't have a quantitative method to measure and evaluate multi-unit cooperative behaviour. Hence, we don't know how much exactly the cooperation have been improved for the two units in this work. Discovering a quantitative scoring schema or develop this schema by ourselves is a crucial work to be included in the future works.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, we implemented state-of-the-art deep reinforcement learning algorithms on a complex and close to reality video game environment, StarCraft II Learning Environment(SC2LE). We focus on achieving better than state-of-the-art performance in both learning speed, average score, and stability in learning. Other than these metrics, we seek a way to increase multi-unit efficiency with the agent.

We have proved that with an MLSH-like structure, an agent can learn stably and achieve better learning speed than the simple baseline structure. We further experimented the multi-unit utilization of this structure with a pre-trained model to control a single unit in Move To Beacon Task. This trial is not successful as the

Furthermore, we proposed a unit selector structure to allow the agent to be able to control multiple units more effectively than the state-of-the-art methods. The proposal is based on the analysis of behaviours of our MLSH-like agent. We practised transfer learning together with the unit selector structure to allow the unit selector to be trained with stability.

5.2 Future Works

SC2LE is open sourced and maintained by one of the most famous and achieving deep learning research organizations in the world, DeepMind. I believe there will be an increasing number of academic research with SC2LE. StarCraft II in video games is like the game of Go in board games. It is always intriguing to be able to take a step forward on problems like these. We have improved the state-of-the-art performance of deep RL agents in some mini-tasks and improved multi-unit utilization of the agent. However, there still exist many

challenges in SC2LE. Such as efficient multi-unit controlling, planning, and information acquiring in imperfect information games. These challenges are also commonly found in game AI. I believe by further research and develop agents in SC2LE. Methods will be found to solve these problems in both SC2LE and other games.

5.2.1 More Complex Mini-tasks

Both mini-tasks used in this work are relatively easy. Move to Beacon task is the easiest task from the officially provided task list. And existing agents are already able to achieve close to human performance with it. Collect Mineral Shard's task is a relatively harder problem involves using multiple agents to solve an NP-hard problem. However, compared with either the full game of StarCraft II or other more complex minitasks these tasks are still too easy. In the future existing complex, minitasks should be tested with the unit selector in tasks with more units and more complex goals.

5.2.2 The Quantification of Multi-unit Cooperative Behaviours

In this work, new behaviours with higher cooperation between the two controlled units were observed. However, there is no quantitative metrics to measure the cooperativeness of multiple RL agents at the moment. Therefore, our observation can only be presented in the form of video or text description.

In the future we want to develop a quantitative rating system to evaluate how an RL agent utilize multiple units efficiently. This rating model should look at the problem from multiple different perspectives. Firstly, it should compare the efficiency of multi-unit agent in the terms of cumulative score with single-unit agents. Secondly, efficiency of multi-unit agent should be compared with human game-play history of the same task. Last but not least, the model can be developed from similar models in game theory and sociology. Which are highly related to our focus of teamwork and multi-unit efficiency.

5.2.3 Playing Full Game of StarCraft II

The ultimate goal of this research is to allow AI agent to play the full game of StarCraft II with equal to or better than human expert performance. However, I see this goal unrealistic to be achieved shortly. For now, the goal should be to compose an AI agent with multiple parts to control each perspective of the game. For example, we could implement with Deep RL a unit selector combined, a mining manager, and a resource manager. These modules can be trained separately first to gain the basic ability of their tasks. Then they could be trained

together to learn to coordinate with other modules. After all pieces of training are complete, we will be able to get a whole module that can do resource management in the game. By making several more modules, it is possible to have an agent to play a full SC2 game with close to the normal human performance.

References

[1] CHRISTOPHER J.C.H. WATKINS, P. D. (1992). Technical note: Q-learning. *Machine Learning*, (8):279–292.

[2] David Silver, Thomas Hubert, J. S. I. A. M. L. A. G. M. L. L. S. D. K. T. G. T. L. K. S. D. H. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*.

[3] Diederik P. Kingma, J. B. (2015). Adam: A method for stochastic optimization. *Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015*.

[4] et al., O. V. (2017). Starcraft ii: A new challenge for reinforcement learning. *DeepMind blog post*.

[5] Kevin Frans, Jonathan Ho, X. C. P. A. J. S. (2017). Meta learning shared hierarchies. *Under review as a conference paper at ICLR 2018*.

[6] P. Henderson*, R. Islam*, P. B. J. P. D. P. and Meger, D. (2017). Deep reinforcement learning that matters. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, 2018*.

[7] Richard S. Sutton, A. G. B. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

[8] Vinod Nair, G. E. H. (2010). Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27 th International Conference on Machine Learning,2010*.

[9] Volodymyr Mnih, Koray Kavukcuoglu, D. S. A. A. R. J. V. M. G. B. A. G. M. R. A. K. F. G. O. S. P. C. B. A. S. I. A. H. K. D. K. D. W. S. L. . D. H. (2015). Human-level control through deep reinforcement learning. *Nature*, (518):529–533.

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, M. M. A. G. T. P. L. T. H. D. S. K. K. (2016). Asynchronous methods for deep reinforcement learning. *arXiv:1602.01783 [cs.LG]*.

[11] Werbos, P. J. (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University.

Appendix A

Research Environment Set-up

The set-up guide is used for Linux users. Since the related source code in this work have only been used in Linux so far.

- Install StarCraft II client for Linux following instructions in the link below. Please do not forgot to install maps for minitasks as well.

  https://github.com/Blizzard/s2client-proto#downloads

- Install PySC2 by typing:

```
pip install s2clientprotocol==1.1
pip install pysc2==1.1
```

- Install dependency packages:

```
pip install absl-py
pip install tensorflow-gpu
```

- Clone the source code of this work from GitHub:

```
git clone https://github.com/yx3110/pysc2-agents_with_mlsh
cd pysc2-agents_with_mlsh
```

- Start Training a baseline agent:

```
python -m main --map=CollectMineralShards
```

- Start Training an MLSH-like agent:

```
python -m main --map=CollectMineralShards
--agent=agents.mlsh_agent.MLSHAgent
```

- Start Training an unit selector agent:

```
python -m main --map=CollectMineralShards
--agent=agents.unit_sel_agent.UnitSelAgent
```

Appendix B

Details of Experiment Hyper-parameters

Parameter Name	Parameter Value	Comment
Learning Rate	5e-4	Learning rate for training
Discount	0.99	Discount rate for future rewards
W	120	Training length for master policy update in MLSH
V	240	Training length for joint update in MLSH
Replay Buffer Size	64	Size of replay buffer
Screen resolution	64	Size of screen image in state representation
Minimap resolution	64	Size of minimap image in state representation
Frame multiplier	8	Number of frames in each timestep
MLSH-like threads	4	Number of training threads in MLSH-like agent
Unit selector threads	8	Number of training threads in unit selector agent
Baseline threads	8	Number of training threads in baseline agent
Optimizer	Adam	The neural network optimizer used for all experiments

Table B.1 Experiment Hyper-parameters

Appendix C

SC2LE Actions and Action parameters

Action Type	Action Parameter Types	Example Value
no_op	[]	[]
move_camera	minimap coordinate	(12,32)
select_point	[type of select, screen coordinate]	[select_unit, (12,32)]
select_rect	[type of select, screen coordinate1,screen coordinate2]	[select_unit, (12,32),(32,12)]
select_unit	[type of select, unit ID]	[single_unit, [0]]
control_group	[type of control group action, group ID]	[set_group, [0]]
select_idle_worker	[type of select]	[select_all]
select_army	[select_add]	[True]
select_warp_gates	[select_add]	[False]
select_larva	[]	[]
unload	[unload ID]	[[3]]
build_queue	[build_queue ID]	[[2]]
cmd_quick	[queued]	[5]]
cmd_screen	[queued, screen coordinates]	[[3], (12,32)]
cmd_minimap	[queued, minimap coordinates]	[[2], (12,32)]
autocast	[]	[]

Table C.1 Categories of Actions and Example Actions in SC2LE