Large Action Space End-To-End Reinforcement Learning Based On StarCraft II

Keyi Yuan, Yintao Xu, Ruiqi Liu, Qi Qin
School of Information Science and Technology
ShanghaiTech University

***Abstract*—In this report, we will use *SC2LE* (StarCraft II Learning Environment) to implement automatically gaming, which is reinforcement learning environment based on the game StarCraft II. Thus, *SC2LE* offers a new and challenging environment for exploring deep reinforcement learning algorithms and architectures. And we will use modified algorithms to make it better. The full game of sc2 is much more challenging. Therefore, we design a new mini-game which conserves as many decision problems as possible with respect to the original full one. New well-designed reward sand an elaborate model are provided More advanced modeling like TQGP(tree-leveled q-value generating process) is implemented in our model to solve the large action space problem.**

I. Introduction

Star craft II is a challenging topic in reinforcement learning. It is a RTS game on PC owned by Blizzard Entertainment. Deep Mind provides a well-designed API and the corresponding baseline models [1]. Pysc2 is more fair and provides similar API to human beings rather than the default API provided by billiard.

The result from baseline model shows that end-to-end RL(A3C) for full game is impractical. In this project, we raise a more practical mini-game which conserves as much mechanics in original game as possible. At the same time, we raise a new model TQGP(tree-leveled q-value generating process) to solve the large action space, which proves to work(we will point out the limitation in the base line model).

II. Mini-game introduction

The full game of star craft II is quite challenging [1]. Therefore, we model a mini-game which per serves as many elements as possible as the full 1-v-1 game.

A. The game rules

In this game, the only goal is to command your troop destroy other player's barracks. The participants are two players: which can be a human being player or a robot.

For the game setting:

- You receive 10 minerals every second, which is different from the practical StarCraft II.
- 50 minerals can be used to train a marine in one's barrack
- everyone has his/her initial troop: 10 marines & 2 medivacs.
- For marines, it can cause damage. And it is the only damage source in this SC2 minigame. And it can be "produced" by mineral.

Fig. 1. The map we design.

- For medivac, it moves faster, but can not attack. And it can heal the injured marines, but it can not be "produced". Therefore, if 2 medivacs are destroyed, your army can not be healed.

Fig. 2. We have 10 marines and 2 medivacs in initial troop.

B. Reward design

As most reinforcement learning problem settings, reward setting is critical and determines the performance of the model.

- each action with errors will get -10 punishment
- to avoid overhead of action, if you idle, you get +1 reward.

- time elapse, each second -3 reward.
- If you win the game, gain the reward of 3,000.
- If you kill an enemy, you gain reward of 50.

III. Recall: Deep Q-value network

Q-learning is a reinforcement learning technique used in machine learning. In class, we have learned that Q value, aka the maximum future reward for a state and action, is the immediate reward plus the maximum future reward for the next state. If you have known the present state s and next state s', based on known transition function $T(s, a, s')$ and reward $R(s, a, s')$, we define the Q value is that:

$$Q(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q(s',a')]$$

where a' is the action chosen when the agent is at the state s'.

Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s' (that may depend on both the previous state s and the selected action), and Q is updated. The core of the algorithm is a simple value iteration update (policy improvement), using the weighted average of the old value and the new information:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma max_{a'} Q(s',a')]$$

where $R(s, a, s')$ is the reward received when moving from the state s_t to the state s', and α is the learning rate ($0 < \alpha \leq 1$). The basic concept to understand here is that the Bellman equation relates states with each other and thus, it relates Action value functions. That helps us iterate over the environment and compute the optimal Values, which in turn give us the optimal Policy.

As the professor says in lecture, this algorithm is a Greedy algorithm, as it always chooses the action with the best value. But what if some action has a very small probability to produce a very large reward? The agent will never get there. This is fixed by adding random exploration. Every once in a while, the agent will perform a random move, without considering the optimal policy. But because we want the algorithm to converge at some point, we lower the probability to take a random action as the game proceeds.

But in this game, we will face to a huge state spaces. Imagine a game with 1000 states and 1000 actions per state. We would need a table of 1 million cells. And that is a very small state space comparing to chess or Go. Also, Q-Learning cant be used in unknown states because it cant infer the Q value of new states from the previous ones. Therefore, we will use **Deep Q-Learning** [2] to solve the problem.

In Deep Q-Learning, we utilize a neural network to approximate the Q value function. It is like Approximate Q-Learning. And using a neural network aims to truncate state space with parameter w, where w is the weight for Q-value $Q(s, a, w)$ when state spaces are too large. The network receives the state as an input (whether is the frame of the current state or a single value) and outputs the Q values for all possible actions. The biggest output is our next action. We can see that we are not constrained to Fully Connected Neural Networks, but we can use Convolutional, Recurrent and whatever else type of model suits our needs.

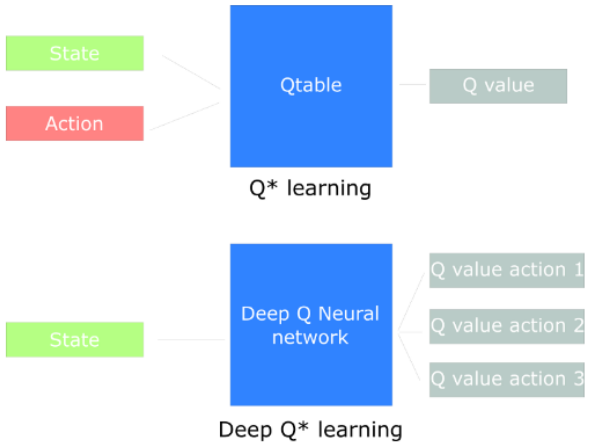

Fig. 3. The difference between Q-Learning and Deep Q-Learning

In this algorithm, the agent holds a memory buffer with all past experiences. Its next action is determined by the maximum output (Q-value) of the network. The loss function is the mean squared error of the predicted Q value and the target Q-value, and it is defined by stochastic gradient descent:

$$loss = (R(s,a,s') + \gamma max_a Q(s',a',w) - Q(s,a,w))^2$$

which is called policy-gradient.

But you can see that this loss function has a problem: the target will change with w. This means it will be hard to fit the target. So we will use two weight: w and w^-, which we will introduce later, to deal with non-stationary . After a certain times, we will update w^- with the current w and w will become a new weight.

The difference between the target and the predicted values is called Temporal Difference Error. Before we train our DQN, we need to address an issue that plays a vital role on how the agent learns to estimate Q Values and this is: **Experience Replay**.

Experience replay is a concept where we help the agent to remember and not forget its previous actions by replaying them. Every once in a while, we sample a batch of previous experiences (which are stored in a buffer) and we feed the network. That way the agent relives its past and improve its memory. Another reason for this task is to force the agent to release himself from oscillation, which occurs due to high correlation between some states and resulting in the same actions over and over. Finally we get make our agent interact with the environment and train him to predict the Q values for each next action.

To remove correlations, build data-set from agent's own experience, and we will use (s, a, r, s') to represent. And we use the sample experiences from data-set and apply update

$$loss = (R(s,a,s') + \gamma max_{a'} Q(s',a',w^-) - Q(s,a,w))^2$$

where w^- is target parameter to deal with non-stationary. And we store transition $(s_t, a_t, r_{r+1}, st+1$ in replay memory D. After that, we optimize MSE between Q-network and Q-Learning targets:

$$L_i(w_i) = (R(s,a,s') + \gamma max_{a'} Q(s', a', w_i^-) - Q(s,a,w_i))^2$$

So the algorithm will be like:

- Given experience consisting of <state, value> pairs: $S = \{< s_1, v_1 >, < s_2, v_2 >, \cdots, < s_T, v_T >\}$
- Repeat: Sample ¡state,value¿ from experience: $< s, v^\pi >\sim S$, and apply stochastic gradient descent update:

$$\triangle w = \alpha(v^\pi - v'(s,w))\nabla_w v'(s,w)$$

As you can see is the exact same process with the Q-table example, with the difference that the next action comes by the DQN prediction and not by the Q-table. As a result, it can be applied to unknown states. This is very useful to us to handle such huge number of states in StarCraft II game.

IV. Design of model

V. Implement DQN to our mini-game

A. Input of state

To implement our aim, we have three kinds of input.
(1) 1-D input, which is an array of size 63, contains:

- current minerals
- count of army
- environment reward
- 20 × 3 selected unit information
- ···

(2) 2-D minimap information, which has the size of 64 × 64, and we have three such 2-D minimaps for:

- camera
- player-relative information [hostile/relative]
- selected unit info
- ···

(3) 2-D screen information, which has the size of 84 × 84, and we have eight such 2-D screen information arrays for:

- visibility
- control
- ···

B. Challenge1: handle huge action space

Using Q-Learning, we expect an action space to be the output, in which contains a lot of Q values as well. Since this is an actions space, for each state, we can have such following action:

- no operator
- move camera ([0∼63],[0∼63])
- ...

And we can reframe it as a tree like that in the following figure.

And there may exists a huge number of Q-values. From this tree, there are $O(b^d)$ Q-values (leaves), where d is the depth of the tree and b is the maximum branch factor. In StarCraft II, we have calculated that for each state, $b = 84 \times 84 = 7056$, $d = 3$. Therefore, there exists $O(\ 3.51 \times 10^{11})$ operations in this tree. The number is too large that it can not be produced end-to-end by network. Therefore, we have our own new idea: **Tree-leveled Q-value Generating Process (TQGP).**

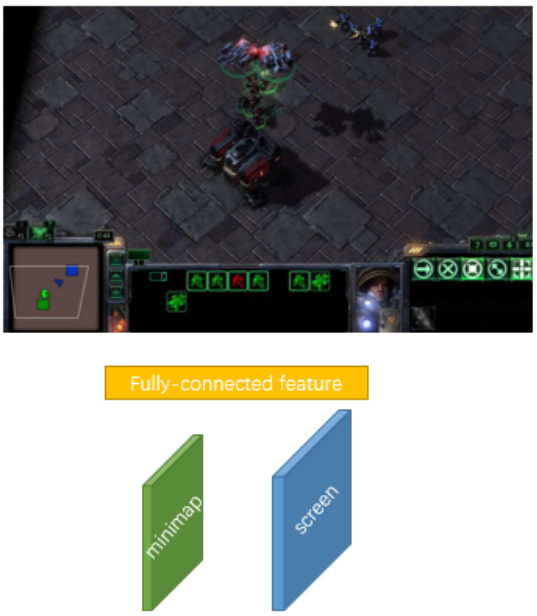

Fig. 4. Three kinds of input.

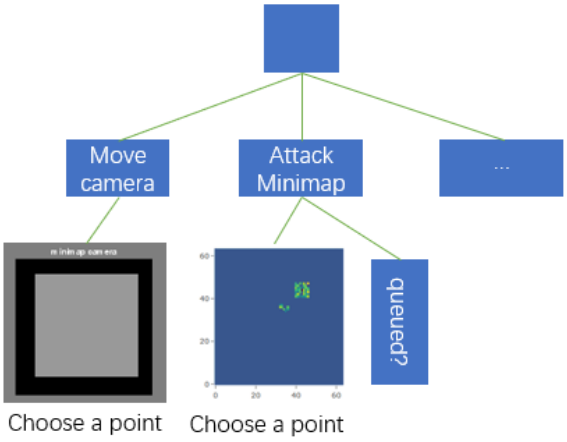

Fig. 5. For each state, we have can such a tree to determine what action to choose for next state.

In TQGP, we have a variable called network outputs importance value (**IV**). It is used to average the same importance but different semantics IV and promote parallel IVs. Therefore, it can be used to promote and average low levels IVs to its parents IVs to compute a multi-leveled action. And the top-level IV in the network is the actual Q-value.

Therefore, we transform each state to importance value (IVs), and the Q-value is made up from a tree contains IVs. We have three operations on TQGP:

- promote

- average
- promote and average

In order to explain the method clearly and vividly, we will use an example to show it.

Suppose we need to compute the Q-value of the operation: Attack to (10,10) in screen and is not queued (since this game allows queued operation, existing a FIFO queue for attack moving).

For now, the task is to find IV of the point we chosen (to attack). Then we need to promote on it. We regrade the importance of choosing all points are the same. The network outputs a $[84 \times 84]$ IVs for choosing a point to move camera, then promote a $[10, 10]$ point IV since we are going compute Q-value of the operation w.r.t to attack [10,10] on the screen. Similarly, we can pick whether to queued by promote operation.

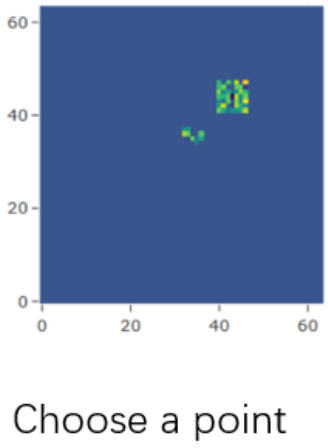

Fig. 6. Choose a point [10,10] to attack.

And then, we need to find IV for the whole operation. Doing average operation, we think that the importance of choosing a point are the same and whether to queue roughly contain similar level of semantics(w.r.t. whether to attack such a general idea). In one word, this operation is to average IV of whether to queue and we know the IV of picking a point.

At last, we need to promote and average for the whole operation. We think that details(picking a point, whether to queue) and general ideas(whether to attack) shares the same importance. So we need to Promote IV of the set of details (gained by average) and to take average with higher level. Their relationship in tree is parent and children.

We average IV of whether to attack minimap with the average of picking a point and whether to queue. Set it as the IV of attack minimap. Now we are at top of the semantic level in tree(except the dummy root). It is the Q-value!

Therefore, we use the following big formula:

$$\frac{(A[10,10] + q[false])/2 + IV_{attack}}{2}$$

Remark: From the big formula, we see the convex combination coefficient of the IV decays exponentially when the depth of the node increases, which indicates the coefficient, primarily defined by the average process, is like a hyper-parameter. Surely, replacing the average unit with a learn-able convex combination is feasible, which intuitively, enables the model to do the trade-off between the more abstract concepts and the down-to-earth details.

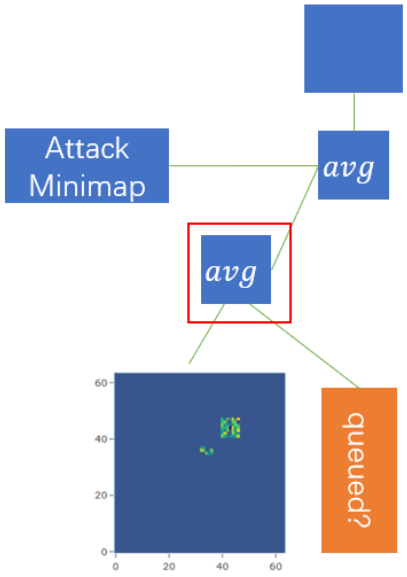

Fig. 7. We need to average IV for each step.

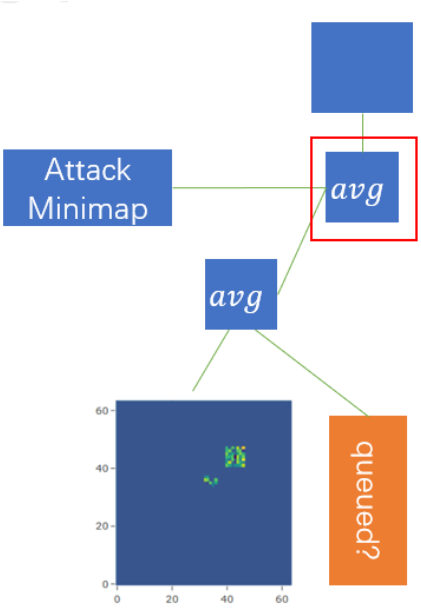

Fig. 8. We need to average IV for whole operation.

If we want to know and pick the maximum Q-value, $max_a Q^*(\phi(s_t), a; \theta)$, then we only need to to take the max Q-value at top level of Q-value tree. But challenge still exists: there are huge amounts of Q-values. We decide to solve it like mini-max methods in class, moving down the max operation.

We use a math trick to guarantee our methods is correct. Since if you want to know $max_{b,d \in R}(ab + cd)$, and a, c is

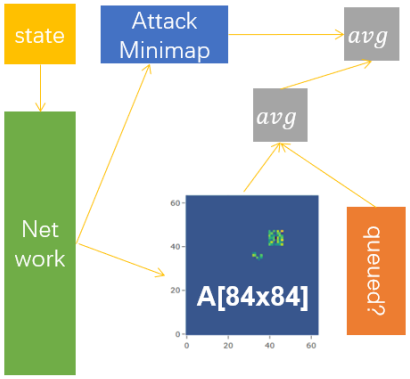

Fig. 9. The process of promote and average. And the *queued* here is an array of size 2.

known to you and both are bigger than 0. Then, given some set A:

$$max_{b,d\in A}(ab+cd) = a\cdot max_{b\in A}(b) + c\cdot max_{d\in A}(d)$$

Since max can transfer on convex combination, then average is a special convex combination with $a = c = \frac{1}{2}$.

Starting from mini-max prune and decision tree, this method has 3 advantages:

- Max only appear on average operator layer and top dummy layer, more efficient.
- Encode prior knowledge of decision into tree architecture.
- Output of network is also skimmed. In practice, 54,652 in our setting.

Remark: Note that the Q-value is the convex combination IVs, which remains the scaling of IVs if IVs scaling are the same(decided by network). In other words, all the Q values are of the same scaling(max and min value), which is of great importance of the convergence of Q-learning. Imagine that initially, the Q value of an action is greater than other one, then it is always selected except the case of random selecting(ϵ greedy method). The process of exploration will be really slow.

The methods above all can make us running this project much more faster. This is a very important method.

C. Challenge2: network architecture design

A memory-less architecture is used in our architecture. In other words, the agent is reflex. We known that it will badly hurt the performance but we have to do. Since the state sequence of the game may be 2000 long, the training cost can be extremely high. Need more advanced modeling trick to avoid this question. For now, we apply a memory-less model.

The input of the network is the state mentioned at section V. The output is the IVs and shapes of them are trivial. If you want more details, you can check our GitHub repository [3].

We also apply an intuitive new(may have been used by someone) layer, the fork layer, which means to fork one value(scalar) to the shape of a 2-d matrix and divided by the size of matrix(the number of the entries). The gradient passed by it is accumulated. The division is to avoid gradient explosion and proves to work in practice.

To shuffle the 1-D information and 2-D information, we design a complicated architecture.

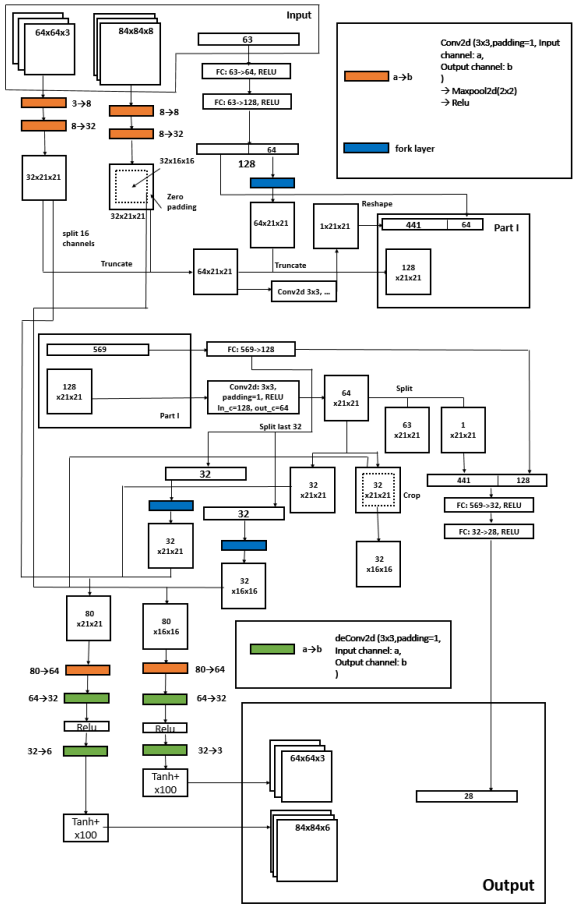

Fig. 10. The whole image of the architecture

VI. Data flow

To sum up, here, I use a figure to show the whole image of the process our agent make decision:

VII. Other tricks

We have tried other tricks to improve the performance our project.

A. Prioritized experience replay

We have this motivation from the paper *Prioritized Experience Replay* [4]. In the prioritized experience replay, we want to take in priority experience where there is a big difference

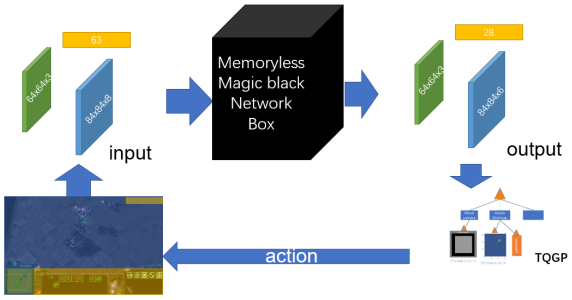

Fig. 11. The whole image of the dataflow

between our prediction and the TD target, since it means that we have a lot to learn about it. We sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D, and we sample by the weight instead of uniformly. And we have modified out implementations:

- Update weight of sample when it is selected to be trained
- Epsilon-greedy method: w.p of ε. It picks sample in replay repository w.p. $\propto$ weight (by some functions); and w.p of $1-\varepsilon$, it still picks sample randomly

And we define the weight as our explore function:

$$weight = \frac{last\ training\ loss}{\#learned\ time + 1}$$

This is a method that can make learning from experience replay more efficient. In our project, this can make the experience replay we have mentioned in DQN part more accurate and faster.

B. Periodic epslion-greedy

This is another way to help us to implement our project. Since we require consistency of operation in playing RTS game. Randomness of a trivial operation may not provide enough heuristics for exploration. In periodic epsilon-greedy, we select a random action a_t with probablity ε. And we have modified out implementations:

- Two mode: sanity mode and random mode
- In sanity mode, it pick action with largest Q value; Otherwise, act randomly.
- With some probability ε, switch to random mode; Otherwise, switch to sanity mode.

Regrettably, this methods seems not good enough for us to use. And we will not recommend you to use this. This is only our own idea.

VIII. Result Analysis

Training takes 2 days, with 450 episodes(game session) on GTX 1080Ti. The model can do 5 operations per seconds which reaches the ordinary human level. We pick a model by competing it with the set of all models from episode 1 to the last. The criterion to pick is to find a model with a higher ternary score.

Ternary score follows the convention by pysc2 [1]. Win gets +1 point, while loss gets -1 point. Otherwise, get 0 point(timeout or other cases).

After experiment, we find that our agent does learning something, with showing 0.7 ternary score over random agent, and 0.2 ternary score over the set of all models from episode 1 to the last. However, the built-in easy hard-code AI still knocks it down by -0.8 ternary score. The result is fairly bad. Here are some reason:

- Our model is memory-less. The inconsistency of operations hurts the performance.
- For our mini-game, there is not much architecture issues. Actually, easy bot performances the same as the hard bot. Commanding troops and attacking may be the advantage of a hard-code bot.

Indeed, it does learn something from its behaviours. From this report, you may not have an intuitive feeling. Video of contest will be added into github repository in the future.

IX. Conclusions

To sum up, our team apply a modified Deep Q learning network model on our mini-game(which is developed by us) of the SC2, which shows poor performance on ternary score. However, it does learn something. Tricks like TQGP(tree-leveled q-value generating process) and fork layers are proved to be useful which is developed and experimented by us. Self-game proves to be an effective way of learning since it gives the models a good start(the opponent is yourself). From this project, we make a down-to-earth and a solid experiment under the topic of reinforcement learning.

References

[1] *StarCraft II: A New Challenge for Reinforcement Learning*, 2017. [Online]. Available: https://arxiv.org/abs/1708.04782
[2] *Playing Atari with Deep Reinforcement Learning*, 2013. [Online]. Available: https://arxiv.org/abs/1312.5602
[3] *github repository*. [Online]. Available: https://github.com/liubai01/AI-StarCraft-II
[4] *Prioritized Experience Replay*, 2015. [Online]. Available: https://arxiv.org/abs/1511.05952