Advanced Starcraft2 Microtasks

A Deep Reinforcement Learning Paper

Nick Lechtenbörger
Institute of Cognitive Science
University
Osnabrueck Germany
nlechtenboer@uos.de

Niklas Schemmer
Institute of Cognitive Science
University
Osnabrueck Germany
nschemmer@uos.de

Johannes Weißen
Institute of Cognitive Science
University
Osnabrueck Germany
jweissen@uos.de

1. Introduction

Since the huge success of the DeepMind presentation [1] of AlphaStar in 2019, StarCraft2 has been a stable testing ground for advanced reinforcement learning algorithms due to its high complexity and the public api [2]. StarCraft2 is a hard game for a reinforcement learning agent to master. Games often have upwards of 50000 actions done therefore providing a sparse reward setting. Additionally, a big part of the game is anticipating the opponents' moves while having imperfect information. Together with a large action space of around 10^8 possible actions [2] an agent will need a lot of training and many smart adaptations to prevail in a full game of StarCraft2.

2. StarCraft2 general terminology

Starcraft2 is a real-time strategy game that is usually played in a 1 vs 1 setting. A player selects one of three races at the beginning of the game and tries to destroy every building of their opponent. The necessary skillets to accomplish this can roughly be separated into two categories "macro" and "micro".

The macro aspect contains the building infrastructure (workers to mine more in-game resources, more buildings and upgrades) or building units designed to attack the opponent. To master the macro aspect of StarCraft2, the capability to plan ahead long term as well as the ability to adapt the own strategy on the fly to the opponents' moves are crucially important.

The micro aspect describes the players' ability to maneuver their own units, select high-priority enemy units, attack, and retreat. The speed of micro is usually measured in actions per minute (apm).

While the macro aspect of the game has been hard solved, as optimal build orders exist for at least 8 minutes for every match-up and unit composition, micro capabilities are nearly unlimited.

3. Motivation

Watching the deep-mind YouTube video [1] back in 2019 was really impressive. A reinforcement learning agent for a game as complex and with as many steps as StarCraft2 could compete with professional players that earn hundreds of thousands of euros playing StarCraft2. A remarkable result. However, there was a big asterisk to their initial result. The agents did not outsmart their opponents, but rather beat them through micro. The agent leveraged its faster and more precise micro to beat the human while having severely worse macro. In a later version [3] this issue was addressed and the current alpha-star variant was tested in the official StarCraft2 ladder, the games competitive, elo-driven game mode. The improved agents now have severe apm restrictions and have to use the in-game camera which levels the playing field between the humans and the agents, while still being able to beat top human players.

Due to the apparent difficulty of creating an agent that is good at macro, we chose a task within the pysc2 environment that favors fast precise actions over long-term planning. We believed that it would be possible to create an agent that has reasonable chances to beat us in a micro competition. Similar to DeepMind [2] we created a specific minigame for the agent to train in using the pysc2 map editor (see figure 4).

4. Task and Environment

We chose a micro task known in the sc2 community as "prism-juggling" [4]. The encounter includes the agent playing as the

“Protoss” race (the selected units in figure 5) and an in-game-bot playing as the “Zerg” race (see top right units of figure 5). Crucially the agent has a flying unit which can pick up ground units, whereas the Zerg bot has slow shooting units, which can only attack ground-only units. Therefore any flying or picked-up units of our agent are safe. Theoretically, it is possible for a Protoss player in this situation to:

```
1. shoot

2. lift its units to escape the slow shots from the opponent

3. drop down its unit all without taking damage while damaging the opponent

repeat
```

Once mastered the agent should be able to defeat an unlimited amount of Zerg units in a row without taking damage. Possible additions to increasing the difficulty of the task would be adding multiple Immortals that need to be managed as well as increasing the number of enemies giving less time to lift the Immortals. We left out these steps due to the poor performance of the agent with only one Immortal, as described in section 6.

4.1 Interaction with the environment

Pysc2 follows the rules of typically gym environments, where the agents interact with the functions step, reset and an initialization function.

The initialization function for example defines the number of players their respective races, screen resolution and game speed.

The score specific to our minigame is given as follows: +5 for every opposing unit killed and -100 if the agent’s ground unit dies which also triggers the reset to avoid useless samples since the agent cannot win or lose any more units.

As the name implies the reset function, retriggers the initialization script within the map file thereby resetting the score and timer while reviving all the units.

The step function works differently from different from normal gym environments. After passing an action id, additional arguments are required. The arguments differ in size shape and type making the action space very complex.

```
0/no_op()

1/move_camera(1/minimap 64, 64])

2/select_point(6/
select_point_act4];

0/screen [84, 84])

3/select_rect(7/select_add [2]; 0/screen [84, 84]; 2/screen2 [84, 84])
```

There can be actions like 0/no_op that accept no arguments at all while arguments 1/move_camera need an additional location for where to move the camera. Actions like 2/select_point can only be executed after certain other actions. The observation returned from the step function is divided into 11 feature layers for the minimap (minimap seen in the bottom left of figure 5), 27 feature layers for the screen and general information like supply, unit count and time passed etc.

5. Implementation

Due to the overall success of Proximal Policy Optimization (PPO) [4] in many different application fields, fast training times, and good generalization capabilities we decided to use PPO as our training algorithm.

Our implementation is based on this pseudo-code (figure 1).

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 1: Pseudocode PPO

After the setup of the environment, the training process repeats for a given amount of episodes.

Next, we need to collect a predefined amount of trajectories through our environment by sampling over the training policy parameterized with the initial training policy parameters. Notably, the environment restarts after a maximum of 120 steps or the death of all “Protoss” ground units. This ensures that the samples always have relevance and agents that refuse to fight do not take up so much time during training.

Following the collection of the rewards, the advantage for each step is calculated from the rewards from each step and the output of the value function for this step as seen here.

```
def finish_trajectory(self, last_value=0):
    path_slice = slice(
        self.trajectory_start_index,self.pointer)
    rewards = np.append(self.reward_buffer[
        path_slice], last_value)
    values = np.append(self.value_buffer
        [path_slice], last_value)

    deltas = rewards[:-1] + self.gamma *
        values[1:] - values[:-1]

    self.advantage_buffer[path_slice]=
    discounted_cumulative_sums(
        deltas, self.gamma * self.lam)
        self.return_buffer[path_slice] =
            discounted_cumulative_sums(
                rewards, self.gamma)[:-1]

        self.trajectory_start_index =
            self.pointer
```

The finish_trajectory method is called once each game is done. The collected game steps are used to calculate the advantage and the return after each step. As the buffer has a fixed size defined before the training, you first need to slice those entries from each of the buffer arrays, that was generated in the last game. From the rewards and their generated critic values, the deltas are calculated (line 4) corresponding to this formula:

$\delta = r_t + \gamma \cdot V(s_{r+1}) \cdot m_t - V(s_t)$.

From the delta values the advantage of each step is calculated (line 5) using the delta and lambda values and summing them with the discounted cumulative sum, according to this formula:

$A_t = \delta + \gamma \cdot \lambda \cdot m_t \cdot A_{t+1}$.

Lastly, the return of each step, excluding the last, is calculated using simply the reward for each step, multiplied by the gamma value (line 8):

$R_t = r_{t+} \gamma$.

The start of the next trajectory is then set to the end of the last one.

While updating the policy the new policy θ needs to stay within θ_{old} +- ε. This ensures that the policy updates are not too big. To achieve this the new policy gets clipped in accordance with [5] and figure 2.

```
min_advantage = tf.where(advantage_buffer > 0,
    (1 + FLAGS.clip_ratio) * advantage_buffer,
    (1 - FLAGS.clip_ratio) * advantage_buffer,)
```

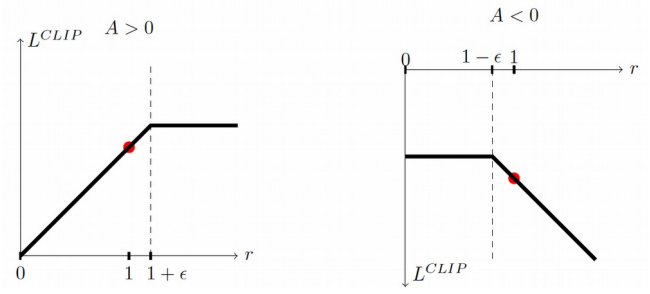

Figure 2: Policy Update Clipping

6. Results and Discussion

As indicated in figure 3 our agent did not manage to learn the desired behavior. Based on the mean reward it received it destroyed a maximum of three opposing units before dying (see reward calculation in section 4). This is equivalent to not giving any inputs and just letting the unit automatically attack the nearest enemy.

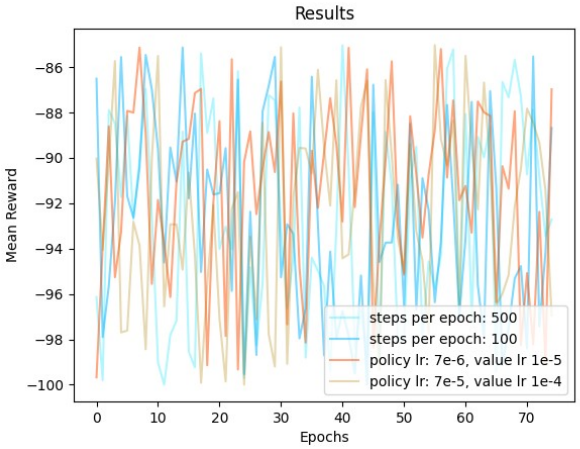

Figure 3: Visualization of Results

We have different ideas as to why our training might have failed but did not come around to test all of them.

First, the agent might not learn due to a faulty implementation of PPO and wrong interaction with the environment. It is possible, that during the implementation the PPO algorithm is fed wrong observations or selects the wrong action due to the separation of spatial and non-spatial actions. The interaction proved to be more complicated than anticipated and the documentation for example about the game's feature layers that completely bloat the observation space is almost non-existent.

This is also part of the next possible reason why the agent did not learn the desired behavior. The observation- and action-space, despite massive downscaling through omitting invalid actions and removing feature layers with useless information for the task at hand as seen in section 5, is still too large and convergence might just take more time.

Bibliography

[1]: DeepMind, AlphaStar: The inside story, 2019, https://www.youtube.com/watch?v=UuhECwm31dM

[2]: Vinyals et al., StarCraft II: A New Challenge for Reinforcement Learning, 2017, arXiv:1708.04782

[3]: Vinyals et al., Grandmaster level in StarCraft II using multi-agent reinforcement learning, 2019, https://www.nature.com/articles/s41586-019-1724-z

[4]: Starcraft2Fan, Immortals Prism Juggling | Dark vs. PartinG on Triton LE, 2021, https://www.youtube.com/watch?v=xaOe-hrDmfM

[5]: Schulman et al., Proximal Policy Optimization Algorithms, 2017, https://arxiv.org/abs/1707.06347v2

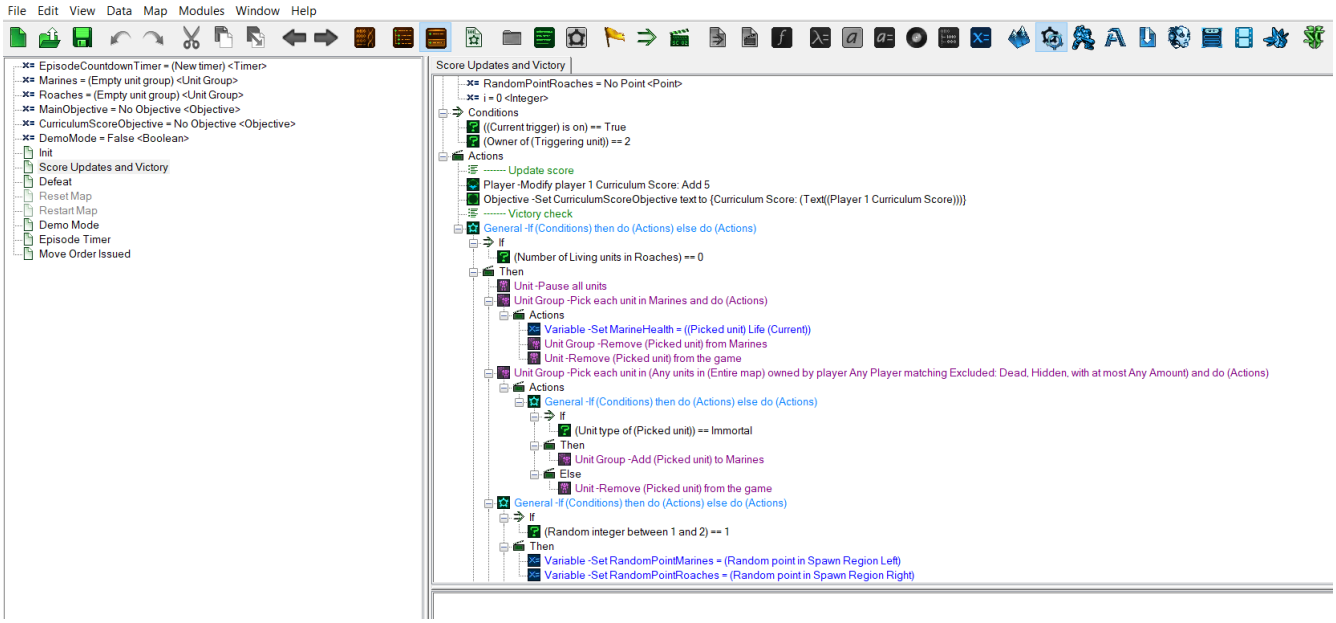

Figure 4: Sc2Map-Editor Example

Figure 5: Custom Minigame for Prism Micro