Multi-agent reinforcement learning on low-end hardware for game AI

Mattéo Sutour, Tom Chollet, Pierre Potel, Albéric De Foucauld, Alexandre Cor

***Abstract*—The SMAC (Starcraft Multi-Agent Challenge) [1] is a challenge were the objective is to win a battle between 8 units in Starcraft. Reinforcement Learning techniques can be used to train the units. After creating the environment, multiple RL methods can be compared and trained to beat the Starcraft II built-in game's AI. The Proximal Policy Optimisation method will be compared to other for this problem.**

Code and ressources can be found at the following link.

I. Introduction

Starcraft is real-time strategy (RTS) computer game using the concept of 'collect-build-conquer'. This work is based on the StarCraft multi-agent challenge (SMAC) [1], a competitive testbed for multi-agent deep reinforcement learning. It specifically adresses the problem of *micro-management* in StarCraft II, a multiplayer strategy game where each player has to build a base and make an army to be the last one remaining on the map. *Micromanagement* refers to the task of moving around each unit on the map and ordering them which enemy unit to attack in order to maximize damage and minimise unit losses. The main challenge on this matter consists in achieving coordinated operations despite a decentralized execution. The scenario 3s5z will be tested in this paper, which is a symmetric scenario opposing two teams composed of 3 ranged unit (Stalkers) and 5 melee units (Zealots).The use of PPO[1] in multi-agent situations has been few and far between in literature[2], but recent findings suggest this learning method boasts competitive performance in multi-agent settings [3]. Indeed Yu & al. state in [3] compared to competitive off-policy methods, PPO often achieves competitive or superior results in both final returns and sample efficiency. Therefore the case in study here is to know whether or not a swarm of reinforcement learning agents can achieve good performance against the game's dedicated scripted AI in a reasonable amount of time and with access to limited computing ressources (mid-range laptops). These results would indicate if reinforcement learning methods could be a viable method for bot and NPC[2] design in low-budget video games (indie games, mobiles games...). Curriculum learning methods, where an agent gradually tackles more and more complex tasks over its learning process, have shown to be efficient at speeding up training in reinforcement learning [4]. We will adopt this approach by sequentially training our agents against the multiple difficulties proposed by StarCraft II's build-in AI (Easy, Medium, Hard) if after a sufficient winrate is achieved.

[1]Proximal Policy Optimisation (PPO) is an architecture ensuring the policy update not to be too large and making the training more stable.

[2]Non-player character (NPC)

II. Background

A. The Reinforcement Learning framework

The RL framework is a discrete-time control process in which an agent interacts in a closed-loop with its environment. At each time t, the agent receives an observation of the state o_t and applies an action at to the environment, in this case the observation is a list of size the number of units, and each unit is described by 5 elements: [Position X, Position Y, Health, Kid of unit (range or melee), Team]. The action generated is a list of size the number of unit in our team and that give an action to do for each unit. As a result of the action, the state of the environment evolves to s_{t+1} and the agent obtain a reward r_{t+1}. Most RL applications consider Markovian processes. A process is Markovian if the evolution of the process only depends on its current state, and does not depend on its history as expressed in Equation 1.

$$\mathbb{P}[X_{t+1}|X_t] = \mathbb{P}[X_{t+1}|X_1, ..., X_t] \tag{1}$$

Given the process is Markovian, the RL process can be described as Markov Decision Process (MDP). A MDP is defined as a tuple of five elements (S;A;P;R;) where:

- S:Finite state space - Discrete or continuous
- A:Finite action space - Discrete or continuous
- P:S x A x $S \xrightarrow{R}$: State transition probability function
- R:S x A x $S \xrightarrow{R}$: Reward function
- $\gamma \in (0, 1]$: Reward discount factor

In an MDP, the state of the system is fully observable, implying that $o_t = s_t$. At time t, $P(s_t; a_t; s_{t+1})$ gives the probability of the state of the system evolving from s_t to s_{t+1}. The set of states, actions and rewards encountered over an episode forms a trajectory τ :

$$\tau = (o_0; a_0; r_1; o_1; a_1; r_2; ...)$$

The value of a trajectory is quantified by the return R(τ), a discounted sum of rewards along τ :

$$R(\tau) = \sum \gamma^k r_k \tag{2}$$

Within the context of the MDP, the goal of a RL agent is to find a policy π which maximizes the expected return. In this work, the policy π is considered to be a stochastic function $\pi(s; a) : S$ x $A \rightarrow [0; 1]$ yielding the probability that a specific action a is chosen given state s .

B. Policy Gradient Methods

Instead of inferring the policy via the value function, policy gradient methods learn the policy directly. The policy is usually parameterized by a neural network with weights θ. The goal then consists of finding the weights θ that maximize the expected return when following the resulting policy π_θ. In that respect, policy gradient methods tackle the RL problem head on, focusing directly on the policy and turning the task into an optimization problem. We can thus define the objective function J:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{+\infty} \gamma^t R_{t+1}\right] \tag{3}$$

Starting from randomly initialized weights, the goal is to find θ^* such that:

$$\theta^* = \arg\max_\theta J(\theta) \tag{4}$$

As the name of the method suggests, the weights are updated ascending the gradient of the objective function J with respect to the weights:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{5}$$

where α is the learning rate. To compute the update increment, the gradient of the objective is thus required. The policy gradient theorem gives an expression of the gradient as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, Q_{\pi_\theta}(s,a)] \tag{6}$$

where $Q_{\pi_\theta}(s,a)$ is the state-action value function. This expression of the gradient allows to compute it in terms of two separate, known quantities: the gradient of the log policy and the Q function, equal to the expected return . Consequently, using a Monte-Carlo approach to estimate Q, i.e the expected return, as the empirical return over a trajectory, we can compute an estimate of the gradient. The REINFORCE algorithm ([5]) depicted in Appendix V-A makes use of this technique; the pseudo code of the algorithm is shown in Appendix.

As a Monte-Carlo method, the REINFORCE algorithm benefits from an unbiased estimation of the gradient, but suffers from a high variance. To reduce this variance while keeping the bias unchanged, a simple approach introduces the concept of the *advantage* of taking an action. An action is said to be *advantageous* if the value of taking that action is higher than the value of being in the current state. The advantage function is defined as follows:

$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s,a) \tag{7}$$

It appears naturally that the value of Q_{π_θ} should be close to V_{π_θ} for a given state and action — unless the action is especially advantageous, meaning that the absolute value of A_{π_θ} should be significantly reduced compared to Q_{π_θ}. Moreover, injecting A_{π_θ} instead of Q_{π_θ} in the right-hand side of (6), we observe that:

$$\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, A_{\pi_\theta}(s,a)] = \nabla_\theta J(\theta) \tag{8}$$

where $d_{\pi_\theta}(s)$ is the distribution of states under π_θ. This equation is derived from the fact that the sum of the policy across all possible actions is 1 and that the gradient of a constant is zero.

Therefore, injected into the computation of the gradient of the objective function, the advantage function does not change the expectation but allows to significantly decrease the values of the increment in (5). This leads to a significantly reduced variance compared to using an estimate of Q in the gradient update. Other techniques exist to further reduce the variance in policy gradients: the actor-critic method addresses this problem by providing a new architecture for the RL agent.

C. Proximal Policy Optimisation

Proximal Policy Optimization is a recent advancement in RL showing remarkable performances. It is an extension of the Trust Region Policy Optimization (TRPO), an approach that ensures that each policy update is not destructive and in the same time not too small — which would lead to extended training. In TRPO, the objective function called surrogate objective and denoted L^{CPI} (for conservative policy iteration) is slightly different from the one of policy gradients — but leads to the identical optimization problem. The goal is to:

$$\underset{\theta}{\text{maximize}}\ L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}((a_t|s_t))}\hat{A}_t\right] \tag{9}$$

$$\text{subject to } \hat{\mathbb{E}}_t\left[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \leq \delta \tag{10}$$

Using the KL[3] constraint is a way to keep the difference between the old and new policy distribution low, hence implementing a *trust region* where the policy lies.

Although TRPO avoids shortcomings of the actor critic methods, the hard KL constraint adds additional overhead to the optimization process and may lead to undesirable training behaviors. PPO changes the objective function in order to include implicitly the constraint on small policy update in the optimization problem. The objective function becomes L^{CLIP}; defining $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}((a_t|s_t))}$, we can write:

$$L^{CLIP} = \hat{\mathbb{E}}_t\left[\min\left(\underbrace{r_t(\theta)\hat{A}_t}_{\text{TRPO objective}}, \underbrace{\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t}_{\text{clipped TRPO objective}}\right)\right] \tag{11}$$

r_t is the ratio between the probability under the old policy and the probability under the current policy to select a specific action in a state. In other words, $r_t > 1$ for a specific action means that it is more likely to be taken under the new policy (compared to under the old policy), and $r_t < 1$ means that it is less likely to be taken under the new policy. More importantly, values of r_t far from 1

[3]The Kullback–Leibler (KL) divergence is a measure of the statistical distance between two probability distributions.

mean that the likelihood of taking the given action varied much during the previous policy update. This is exactly what TRPO and PPO seek to avoid: big changes of the policy in a single update that could damage the policy learned so far.

As a reminder, the goal of the policy update is to move θ so that to increase L^{CLIP}. If the advantage is positive, the way to increase L^{CLIP} is to increase the TRPO objective (see (11)), therefore to increase r_t. However, as we take the minimum of the clipped and unclipped objectives, L^{CLIP} cannot exceed $\hat{\mathbb{E}}_t[(1+\epsilon)\hat{A}_t]$. This ceiling value prevents the incentive to increase r_t above $1+\epsilon$. Conversely, for actions with a negative advantage, the clip prevents decreasing r_t below $1-\epsilon$. In short, the PPO objective is such that the new policy does not benefit from changing too much from the previous one.

The PPO algorithm 2 is divided into 2 threads: the first one collects sequences of episodes and computes the advantage estimate. Once a batch of episodes is collected, a second thread runs gradient ascent on the policy network using the PPO objective.

III. Methodology/Approach

To create the environment, two main propositions have been tried. The first one is based on an idea from Sentdex [6]. His work focuses on creating an AI for the real Starcraft II Game by training a Reinforcement Learning model on the mini-map from the game. Looking directly at the footage from the game would be too difficult as the model would need to learn what everything on the screen means. The environment is a RGB map of the field as shown in figure 1.

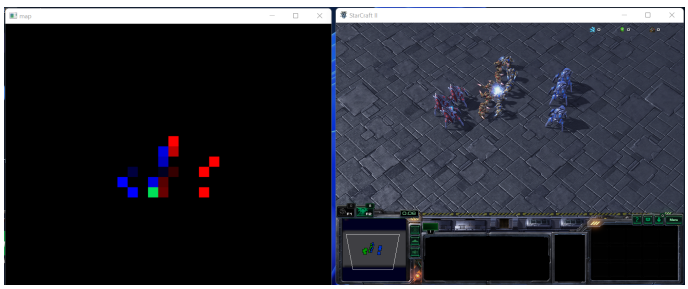

Figure 1: Visualization of the first environment representation and the real game.

In this environment, the red points represent the enemy units and the blue points our units. The green point is the agent that has to chose an action. And the brightness of the points depends on the health bar of the corresponding unit.
As the problem we are trying to solve is dealing with multi-agent Reinforcement Learning, the main issue for this environment is that the model still has to understand what unit it is controlling at each step. And this map observation is not the best representation when one is trying to make micro-decisions. It would have been indeed very useful if the model had to predict macro-decision as it is a good overview of the playing area but in our situation it might not be the best. Moreover to create team strategies controlling one unit at a time is not the most efficient technique either.
That's why we decided to use another representation of the environment instead. That new representation is in the form of a matrix with each row being a state vector for each agent. As there are 8 ally units and 8 enemy units, there are 16 rows in our environment representation. Each unit's state vector has 5 fields : the x and y positions on the playing area, the health of the unit, its team (ally or enemy) and its type.
Now the model can choose the best action for each ally unit at each step, knowing all the units position, health, teams and type. In particular, we are using 2 types of units, the Zealots and the Stalkers. The Zealots attack hand-to-hand, whereas the Stalkers attack at a distance. This way, the model can learn to take different actions and different behaviors depending on the unit type, and thus can create more elaborate strategies.
With this environment the model is going to chose for each unit, sequentially, an action between 5 actions possible: Moving Left, Right, Up, Down or Attacking the closest enemy.

The aim of the reward function is to foster the model to learn patterns leading to victories. A victory is analysed here as a game which ends where the cumulative health of all the enemy units is down to zero, being reduced at each step taken by the model. From this modelling choice, the goal of this learning agent is then to reduce the cumulative health of all enemies when taking an action, meaning that the cumulative health should be lower at current step than the one at previous action step. This explain the choice of the following form for reward function for this project's model:

$$r_t = \sum_{Ennemy\ units} health^{<t-1>} - \sum_{Ennemy\ units} health^{<t>} \tag{12}$$

As every unit is given a *shield* - which is a extra health bonus that has to be destroyed before the proper health of a unit could be damage[4]- and as the ultimate goal is the win rather than just the reduction of ennemies'health, a specific reward bonus is given when the game ends by a victory. A penalty is also given when it ends by a loss. Therefore, the reward function used in this report is :

$$r_t = \sum_{Ennemy\ units} (h+s)^{<t-1>} - (h+s)^{<t>} + \begin{cases} +100 & \text{if } win \\ -100 & \text{if } loss \end{cases} \tag{13}$$

with s for the shield status and s for the health of an unit.

Last, to evaluate the performance of our model, allowing to compare with state-of-the-art models [6],

[4]The particularity of a shield rather than just adding more health is that, unlike a unit's health, a shield could regenerate over the game

the evaluation metrics would be the test win-rate of the learning agent, which corresponds to percentage of wins against a bot whose difficulty is set to *Very Hard*. However, regarding the weight of the computation of the training even for a difficulty setted to *Easy*, the metric used in practice is the win rate against *Easy* bots.An additionnal metric used is the evolution of the remaining health of the units after every iteration. This other metric help seeing the progression of the training even when their is not win yet.

IV. Results and Discussion

We now want to train our Reinforcement Learning model for that multi-agent fight scenario in StarCraft II. All the implementations for the environment, agents and rewards are successfully achieved. We are using the stable baseline implementation of the PPO algorithm to train our model. After unsuccessful experiments exploring other hyper-parameters set-up, the default values implemented by stable baseline 3 were used. These hyper parameters for the PPO method are depicted in the table below:

Policy model	CNN Policy [5]	Batch size	64
Learning rate	3x10^4	#Epochs	10
#steps[6]	2048	γ	0.99[7]

Table I: Hyper-parameters of the PPO implementation (Default from SB3)

In our first results, the model only returned one discrete value corresponding to the action taken knowing the current state of the environment. That result was not appropriate, as we are trying to solve a problem with multiple agents. Thus, all the ally units were taking the same action at each step, which was not the aimed behavior. Instead we modified the action space to be multi-discrete, giving which decision that has to make each of the several agents.

Different training process have been computed with different reward function and environment description. The first runs was using a simple yet efficient reward function that was giving a positive reward when an agent was successfully hitting an enemy. This was implemented to help the algorithm understanding how to win. In addition, a bonus reward of +-100 was given at the end of the game as described in equation (13). The results of the training is shown in figure 2. This graph is showing the average remaining health of the units after a battle. This remaining health can be both positive if our team wins or negative if it loses. The metric is computed over 20 battles every 100 iterations.

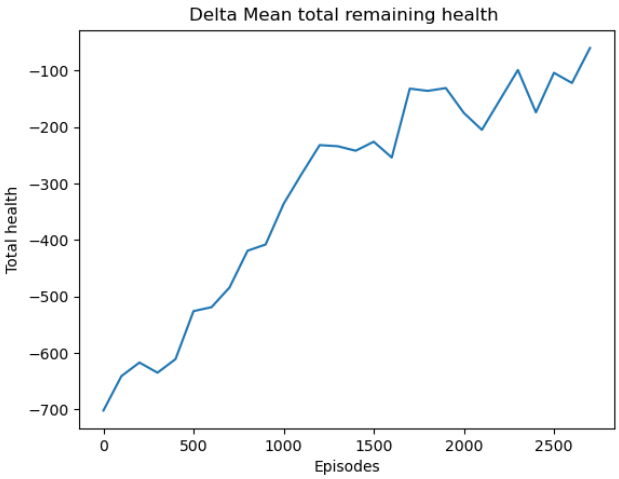

Figure 2: Evolution of the end health total during training.

This first idea gave a final win rate of **23%** against the easy bot after 2300 iterations (which is about 8h of training). The idea was simple and needed to be improved.

After implementing the reward function described is equation (13) and training the model with this idea the results were as followed. The metric showed in figure 3 is the moving average over 10 battles of the final total health and shield for the enemy team. A win is a 0 with this metric.

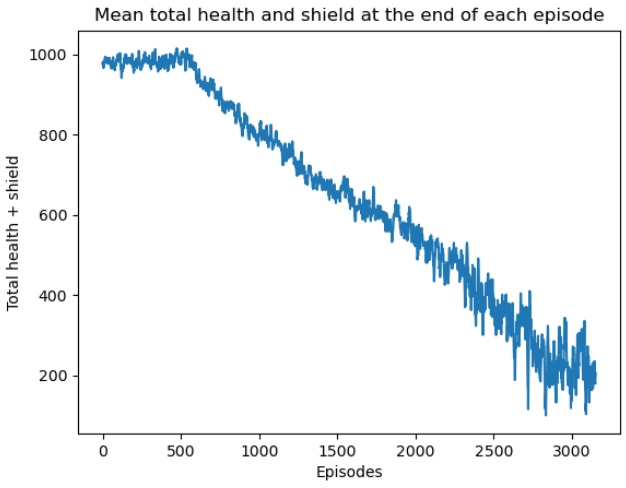

Figure 3: Evolution of the end total health and shield of the enemy team during training.

Here, after about 3000 iterations, the win rate the easy bot is **28%**. The training started difficulty with over 500 iteration of stagnation. The previously used reward the was just giving a positive reward to a strike was more effective in the beginning. On the other hand, the final results are slightly better with this idea regarding the win rate for a similar computing time. Although the 2 ideas

[5] As the observation state is provided as a 2D matrix, the policy network chosen is 'CnnPolicy'.In this configuration, the architecture chosen is called the "Nature CNN" and is used for the Actor and the Critic. However, as the observation matrix is composed of 1D arrays of observations of each agent and as we wanted to prevent an agent to access the information of another agent, we chose to use the 'MlpPolicy' network after linearising the observation matrix.

[6] The number of steps to run for each environment per update

[7] The aim is to maximise the return of an episode rather than the instant reward

are very similar as they are praising attack to help the model improve, the second one which is more descriptive of the environment is giving slightly better results in the end.

With this model performing decently against the *Easy* bots, a strategy used to continue the training is to perform some curriculum learning. Changing the difficulty to *Medium* then *Hard* when the win rate is sufficient allows the model to train for gradually more complex tasks.

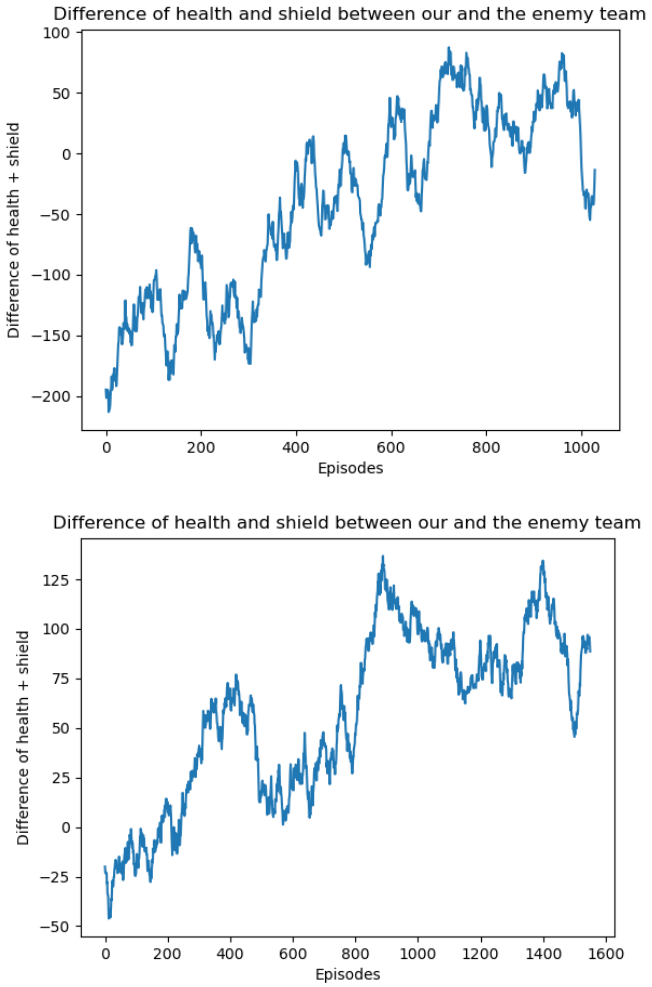

Figure 4: Evolution of the end total health and shield during training for Medium and Hard mode.

The win rate after a 1000 more epoch against the *Medium* bots is **56%** and after 1600 more episodes against the *Hard* bots the win rate is **71%**. Their was no significant drop in the performance of each models when switching to *Medium* and *Hard* mode. The bots were having similar policy so the training continue smoothly.

Some unsuccessful training have also been performed. One first implementation of the idea described just above was not taking into account the shield of the units but only the health. Because the unit have to get their shield destroyed before losing health, the reward was always 0 in the beginning of the training and the model was not able to understand how to win a game. The win rate for this implementation was **0%**.

Another idea was to give to the units the possibility of choosing a target by increasing the decision space. A simple implementation tested was to let the model picking a number between 1 and 12. With an action between 1 and 8, the unit was trying to attack the corresponding enemy unit. For an action between 9 and 12 it was moving as previously implemented. The training failed and it didn't learn correctly. The issue with this idea was a wrongly designed decision space. It was indeed unbalanced, giving to much possibilities for attacking. Moreover, a lot of attacks failed to hit an enemy as the target was too far away from the unit trying to attack. The win rate for this implementation was also **0%**.

Implementing a reward function that takes into account the current health and shield of both our team and the enemy team didn't worked either. The reward was implemented as follows:

$$r_t = \sum_{Ennemy\ units} (h+s)^{<t-1>} - (h+s)^{<t>} - \sum_{Our\ units} (h+s)^{<t-1>} - (h+s)_{<t>} + \begin{cases} +100 & \text{if } win \\ -100 & \text{if } loss \end{cases} \tag{14}$$

This was created to help the model maximizing the damage dealt and minimizing the damage taken. The learning process was extremely slow though. Indeed, the reward function was to often outputting a value close to 0 and even though it was learning the lack of information was causing extreme slowness.

V. Conclusions

Multiple tentative have been tried to get the best results possible for this problem. What can be highlighted is the importance of the problem definition as a lot of dead ends have been encountered during the experimentation phase. Reward implementation ideas that seemed more descriptive of the environment sometime raised unexpected issues and simple ideas gave good results. The difficulty of the problem implied long training process to get interesting results. This might also be related to the environment representation that could be difficult to understand for the model. If our team had enough time to further experiment, this would be our main topic. Finally, a win rate of **71%** against the Hard bots is very positive final metric.

References

[1] M. Samvelyan, T. Rashid, C. S. de Witt, *et al.*, "The StarCraft Multi-Agent Challenge," *CoRR*, vol. abs/1902.04043, 2019.

[2] B. Baker, I. Kanitscheider, T. Markov, *et al.*, *Emergent tool use from multi-agent autocurricula*, 2019. DOI: 10.48550/ARXIV.1909.07528. [Online]. Available: https://arxiv.org/abs/1909.07528.

[3] C. Yu, A. Velu, E. Vinitsky, *et al.*, *The surprising effectiveness of ppo in cooperative, multi-agent games*, 2021. DOI: 10.48550/ARXIV.2103.01955. [Online]. Available: https://arxiv.org/abs/2103.01955.

[4] P. S. Sanmit Narvekar, "Learning curriculum policies for reinforcement learning," 2018.

[5] Sutton and Barto, *Reinforcement learning*, MIT Press, 2020.

[6] Sentdex, "A. i. learns to play starcraft 2," 2022. [Online]. Available: https://github.com/Sentdex/SC2RL.

Appendix

A. REINFORCE Algorithm

Algorithm 1 REINFORCE

Initialize θ arbitrarily
for each trajectory $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
  for $t = 1$ to $T-1$ **do**
    $g_t \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t}\, r_{k+1}$ ▷ compute return from t onwards
    $\theta \leftarrow \theta + \alpha\, \nabla_\theta \log \pi_\theta(s_t, a_t)\; g_t$ ▷ update θ with gradient ascent
  end for
end for
return θ

B. PPO Algorithm

Algorithm 2 PPO

Initialize θ, w and s arbitrarily
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectory by running π_θ in the environment
  Compute advantage estimates $\hat{A}_t$ based on current value function $V_{critic}(s_t, w)$
  Update policy parameters θ maximizing PPO objective: Eq. (11)
  Update critic parameters w minimizing MSE loss
end for