RL team project final report

RL project final report: Gradual Restriction with SC2

201911025 김승하

201911050 도담록

201911185 추성재

Introduction

The artificial intelligence (AI) for StarCraft 2 has demonstrated impressive performance. However, a debate persists concerning whether its success came from an optimal reinforcement learning strategy or simply from high Actions Per Minute (APM) facilitated by machine micro-control. During the official match between AlphaStar and Mana, AlphaStar exhibited a momentary peak APM of 1300 at playtime 11 min 40 sec into the 4th game. Meanwhile, Eris, the first AI model in the StarCraft 2 AI Arena, maintains an APM exceeding 20,000. In contrast, human players typically sustain an average APM between 200 and 400, with occasional instances of surpassing 800 APM.

A simplistic solution to address this disparity involves imposing a limit on APM during training and evaluating. Nevertheless, a potentially more efficient strategy could start initiating learning without APM restrictions, subsequently introducing limitations incrementally to attain a predetermined APM level. The primary objective of this study is to contrast the efficacy of these two methodologies.

Backgrounds

Efforts to apply reinforcement learning to StarCraft II have been ongoing, and pysc2 has been used as an API for such research. However, pysc2 presents a challenging learning environment due to the use of low-level human interface as the action space, resulting in thousands of actions based on clicking positions on the screen and hundreds of unique actions for each unit in StarCraft II.

Sun et al. (2018) applied the off-the-shelf RL algorithm PPO to a StarCraft II reinforcement learning agent called TStarBot1 by utilizing macro actions, which are bundles of pysc2 actions grouped by functionality, as the action space. The macro actions of TStarBot1 demonstrate efficiency in three aspects:

1. Hierarchical nature:

To achieve victory in StarCraft II, a real-time strategy (RTS) game, training must occur at abstract levels encompassing global strategies, local tactics, and micro-execution levels.

2. Hard game rule learning:

To employ strategies effectively, information about the game's systems and rules, such as technological advancements and advantages/disadvantages of attack methods, needs to be learned explicitly or implicitly.

3. Uneconomical learning for trivial decision factors:

In StarCraft II gameplay, it is more efficient to learn factors that have a significant impact on victory rather than wasting time learning elements that are less relevant to winning.

Due to the usage of actual human actions such as mouse clicks and keyboard button presses as the action space in pysc2, hundreds of hotkeys and thousands of mouse-click coordinates on the screen form an extensive action space. Consequently, learning abstract

victory strategies and game rules takes a considerable amount of time. However, TStarBot's macro actions wrap individual actions for efficiency, enabling faster learning.

In the case of AlphaStar utilizing pysc2 with its complex action space, the AlphaStar Supervised model, trained through supervised learning, was further reinforced through 50 days of reinforcement learning to obtain the AlphaStar final model. In contrast, TStarBot, with only 1-2 days of training, achieved an 80%-win rate against the "insane" difficulty level in StarCraft II. Although AlphaStar performs better when all the trains end, TStarBot trains quite faster than AlphaStar.

Methodology

The training of the Tstarbot1 Proximal Policy Optimization (PPO) agent was conducted in alignment with Starcraft2 (SC2) artificial intelligence (AI) models. This method corresponds to the training procedure detailed by Tstarbot, which encompassed simultaneous training with the same number of AI with all levels. However, the inherent inefficiencies of this method are the potential absence of meaningful rewards when the AI level is too high or too low. An AI level that is "too low" could consistently result in a win which means reward = 1 always and conversely, an AI level that is too high could persistently yield a loss, which means reward = -1 while all training.

To boost the learning process, we matched the agent’s level to the AI's level. The initial stage commenced with an AI level of 1, signifying the easiest level. The AI level was escalated to the succeeding level whenever the win rate exceeded 0.5 for a span of 100 games.

In practical scenarios, the execution of actions at a rapid pace within a short duration is unfeasible. Previous research has accounted for Actions Per Minute (APM), but this measure fails to concern the difference between actions that are possible and those that are not. While a continuous 400 APM might be achievable, an initial 1200 APM for the first 20

seconds followed by 0 APM for the other 40 seconds is unrealistic. Thus, we incorporated a short-term restriction, characterized by enforcing noop if $n_k < 100\alpha$ for all k and $n_{k+1} = \begin{cases} \max(n_k - 1, 0) & \mathrm{n_k} \geq 100\alpha \\ n_k + 1 & n_k < 100\alpha \end{cases}$. k is a step when agent performs action and n_k is an k-th step indicator that counts the number of actions without noop between a short-term. noop means an action that the agent does nothing. α is the limit number of actions. In situations where $n_k \geq 100\alpha$, the agent is required to perform a noop action. indicating a rest period to reduce n_k. The value of α was set to 0.2, which means that the agent can perform a maximum of 20 actions in the period. For a higher volume of actions, the agent must rest to decrease n_k.

However, the immediate imposition of restrictions on a pre-trained model could lead to failure. A gradual application of restrictions might allow the model to adapt accordingly. We therefore introduced a comparative study involving two models. The first model was imposed to a hard restriction α=0.2 directly, whereas the second, our proposed model, was imposed a soft restriction, transitioning gradually from α=1 to α=0.2. Both models were subsequently retrained.

Results

Random model

A random agent refers to an agent that performs actions randomly selected from the action space provided in TStarBot1, which are executable at the current observation. Consequently, the random agent tends to choose currently producible units rather than strategically accumulating resources to meet technical requirements. Typically, the random agent prioritizes constructing the least expensive buildings—such as extractors, evolution chambers, and spore crawlers—which do not require specific technical prerequisites. Eventually, the random agent expends resources on building the cheapest structures, such as

spore crawlers and spine crawlers, and fails to build a comprehensive military force, leading to its defeat.

Figure 1: Random Agent vs. SC2 Bot Difficulty 7

PPO trained model

We trained the TStarBot1 PPO algorithm model with 250000 games. The learner agent started training after the actor agent performed 2500 games and saved checkpoints at 50000, 100000, 150000, 200000, and 250000 games.

This is a picture showing the agent's gameplay performance according to the learning result. In Checkpoint 50000, agents produce units with low consumption resources and no technical requirements, showing behavior similar to random agents. On the other hand, agents at Checkpoint 250000 show a strategy similar to the commonly known Roach and Ravager rush build order in PvP (Person vs. Person) competitions. With PPO algorithm, we obtained an agent to defeat StarCraft 2 bot difficulty 7.

Figure 2 (Left): PPO Agent Checkpoints_50000 vs. SC2 Bot Difficulty 7

Figure 3 (Right): PPO Agent Checkpoints_250000 vs. SC2 Bot Difficulty 7

Figure 4: Abnormal behavior of SC2 Bot Difficulty 8, 9, and A

In our test maps (Simple64), abnormal behavior emerges in the StarCraft 2 AI bot at difficulties above level 8. Beyond level 7, the AI bot gains all vision, which means that AI ignores the entire battlefield's fog and knows everything, having full observation, as a measure to increase the difficulty level. Given that our test map is relatively small(64×64 *while ladder map start from* 128×128, *which is* 4 *times bigger than our map*), the AI's workers make dumb action due to this vision feature, leading to incidents where they ventured into our base and suicide(Image Above). Since the map is too small, it seems that the enemy misinterprets our base's resources as enemy's resources, so they come to gather

resources. Therefore, since map cheaters do not work well, we decided to exclude levels 8, 9, and A, where vision is activated, from the evaluation process.

Restricted trained model

We re-trained the above pre-trained model, with hard α restriction and soft α restriction. We trained them 30,000 games each model, in soft restriction α=1 on the first game and gradually reduce α, in the 25,000th game We trained soft restriction with α=0.2, same as Hard restriction. In 25,000~30,000th game, both soft and hard alpha restriction have α=0.2.

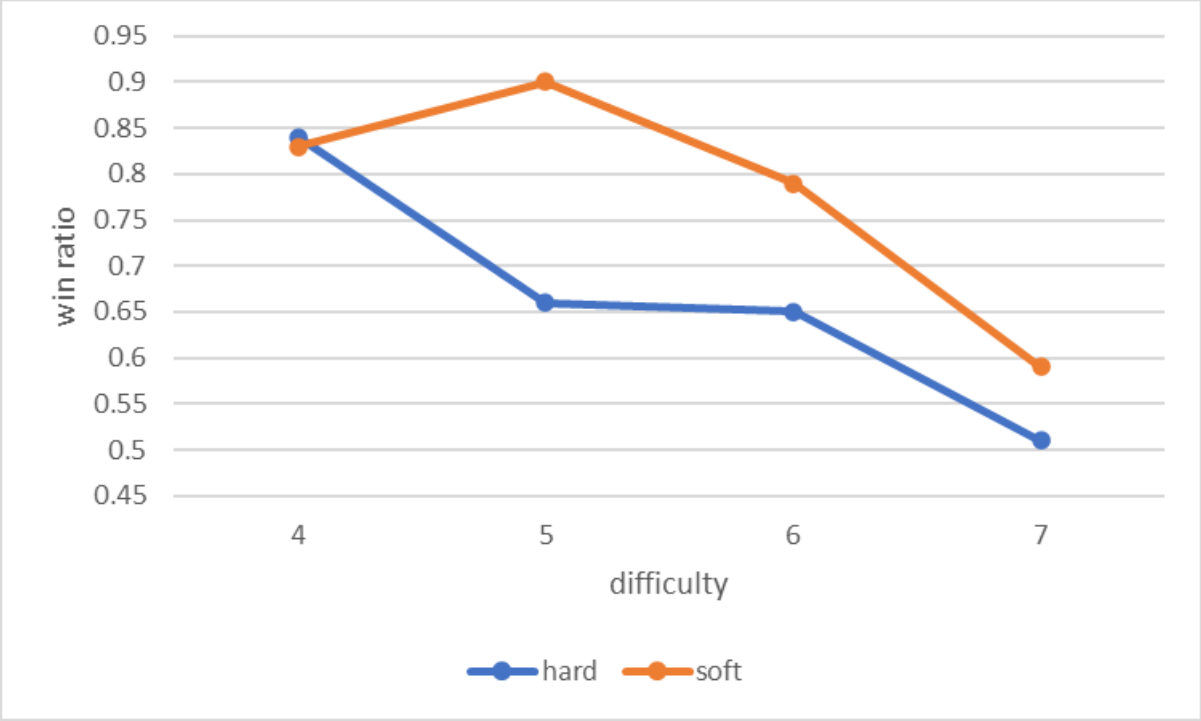

Figure 5: Win ratio of Hard restricted Agent and Soft restricted Agent per Difficulty of SC2 AI Bot

The figure above demonstrates the win rate of models trained using soft restriction and hard restriction, with a restriction corresponding to α=0.2 applied in evaluate.py, across different difficulty levels. As per the observed win rates for both soft and hard restriction, the

model implementing soft restriction consistently achieved higher scores across all difficulty tiers compared to the model implementing hard restriction.

We initially hypothesized that applying a hard restriction from the outset would potentially impede proper training, thus leading to lower win rates. Furthermore, we anticipated that applying a soft restriction, which entails minimal limitations initially, would yield a higher win rate than a hard restriction, due to lesser degradation in training attributable to limitations. The results presented in the figure align with our original hypotheses.

During the training process of the hard restrict and soft restrict agents, we measured the action ratio, defined as the proportion of instances in which the actor performed an actual action rather than a 'no-operation' (noop) in the environment. The table below presents the results obtained when the trained (30,000 games) models competed against an AI with difficulty level 7.

Checkpoint-30000	with restriction	without restriction
action ratio(hard)	0.48	0.73
action ratio(soft)	0.49	0.75

Table 1: Action ratio of Hard restricted Agent and Soft restricted Agent at with restriction and without restriction

Observing the changes in the action ratio, it is evident that the number of actions performed increases when there are no restrictions in place.

Discussion

TStarBot1's APM was not high enough to impose restrictions. TStarBot1 we trained did not have an APM faster than a high-level programmer, with 100 wrapped actions per

minute, which means 200-300 APM(wrapped actions have few actions). This is because TStarbot1 is based on macro actions. Additionally, the training time was shorter than the time conducted by Sun et al. (2018), and the collected data was not as extensive compared to the computing power used by Sun et al. (2018). As a result, the models completed training without using sufficiently high APM, indicating lower APM than what we expected.

This algorithm(gradually changing environment) suggests that when applying a pre-trained model to a different environment, if the changes in the environment can be represented as a continuous metric, it is possible to obtain a model that adapts more rapidly by providing gradual changes for training. For example, when adapting the AI of a robot trained in a gravity in earth environment to a zero-gravity state, it might learn more faster if the gravity is gradually reduced during training, rather than eliminating gravity entirely and retraining in one time.

To impose restrictions, we trained the agent by substituting the agent's actions with a 'no-operation' (no-op) in situations where the restrictions were enforced. Consequently, the agent exhibited an action ratio of 0.5 with restrictions, while an action ratio greater than 0.7 was observed in the environment restriction removed. Our action substitution method was effectively trained to operate under restrictive conditions, and it also was capable of functioning in unrestricted situations.

However, in a restrictive environment, there is a probability of issuing the same action more frequently, given the assumption that actions may not be reflected from this step. Therefore, the agent may exhibit unpredictable behavior when the restrictions are removed.

This action substitution restriction approach differs from a reward restriction method where the output of the action directly influences the reward. If an agent is trained using a reward restriction method where the output of an action directly influences the reward, the agent will behave as restrictions are still in place even when these restrictions are removed.

Future comparisons between the action restriction method and the reward restriction method could explain which approach is much effective under varying circumstances.

Roles in project

All members of our team equitably participated in various tasks ranging from topic selection, proposal of implementation ideas, code development, to the collation of model testing results. However, certain roles were divided among team members or were assigned with a greater emphasis. These responsibilities are outlined here.

김승하: presentation, supercomputer server environment setting

도담록: Abstract algorithm and main idea proposal

추성재: troubleshooting and environment(server, TStarbot1) setting

Place where we modified the code

Modified version of TStartbot1 is uploaded on https://github.com/saychuwho/TStarBot1. This code includes our modification for research and to solve compatibility issues.

We modified “start_actor” in “sc2learner/bin/train_ppo.py” to set the difficulty of SC2 increased gradually.

```
def start_actor():
    policy = {'lstm': LstmPolicy,
            'mlp': MlpPolicy}[FLAGS.policy]
    tf_config(ncpu=2)
    random.seed(time.time())
```

```
difficulty = '1'
#difficulty = random.choice(FLAGS.difficulties.split(','))
game_seed =    random.randint(0, 2**32 - 1)
print("Game Seed: %d Difficulty: %s" % (game_seed, difficulty))
env = create_env(difficulty, game_seed)
actor = PPOActor(env=env,
                                policy=policy,
                                unroll_length=FLAGS.unroll_length,
                                gamma=FLAGS.discount_gamma,
                                lam=FLAGS.lambda_return,
                                learner_ip=FLAGS.learner_ip,
                                port_A=FLAGS.port_A,
                                port_B=FLAGS.port_B,
                                actorID=FLAGS.actorID,
                                softAlpha=FLAGS.soft)

# modified code start
actor.update_difficulty(difficulty)

result = actor.run()
env.close()
#ddr revised
while True:
    if result==1:
        difficulty = str(int(difficulty)+1)
    elif result == -1 and int(difficulty) != 1:
        difficulty = str(int(difficulty)-1)
    game_seed =    random.randint(0, 2**32 - 1)
    print("Game Seed: %d Difficulty: %s" % (game_seed, difficulty))
    envr = create_env(difficulty, game_seed)
```

```
    envr.reset()
    actor._upgrade = False
    actor._downgrade = False
    actor._env = envr
    #actor._env.env.env._difficulty = difficulty
    #actor._env.env.env._diffchange()
    actor.update_difficulty(difficulty)
    result = actor.run()
    envr.close()
# modified code end
```

We modified “nstep_rollout” method in PPOActor inside “sc2learner/agent/ppo_agent.py” to support our restriction.

```
        # modified code start
        # restrict action
        if self._psudoAPM > 100*self._alpha:
            action[0] = 0
        # modified by CHU
        self._total_action_counter += 1
        if action[0] == 0:
            self._noop_counter += 1
        if action[0] == 0:
            self._psudoAPM = max(0, self._psudoAPM - 1)
        else:
            self._psudoAPM += 1
        # modified code end
```

Also in “nstep_rollout”, we modified code to support soft alpha, print action ratio of actions, and difficulty changing.

```
            # modified code start
             #DDR revised
             #revised. here to add apm restriction reward.
             self._ratio = (self._total_action_counter - self._noop_counter) / self._total_action_counter
             self._ratio_list[self._count] = self._ratio
             self._mean_ratio = np.mean(self._ratio_list)
             self._count += 1
             self._wincount += reward
             if self._count%100==0:
                print(self._wincount)
                if(self._wincount>=0.2):
                    self._upgrade = True
                elif(self._wincount<=-0.2):
                    self._downgrade = True
                self._winrate = (1 + self._wincount) * 0.5
                self._count = 0
                self._n += 1
                if self._softalpha:
                    self._alpha = max(0.2, 1 - 0.02*self._n)
                self._wincount = 0
                if not os.path.exists(self._file_output_path):
                    with open(self._file_output_path, "w") as output_file:
                        output_file.write(str(self._winrate)+" / "+str(self._mean_ratio)+" / "+str(self._difficulty)+"\n")
                else:
                    with open(self._file_output_path, "a") as output_file:
                        output_file.write(str(self._winrate)+" / "+str(self._mean_ratio)+" / "+str(self._difficulty)+"\n")
            # reward = reward - self._ratio * self._alpha * 2
```

```
            # reward = max(reward, -1) # no need to cliping
            self._total_action_counter = 0
            self._noop_counter = 0

            #DDR revise end
            # modified code end
```

In “act” method in PPOAgent in “sc2learner/agent/ppo_agent.py”, we modified code to support restriction when evaluating model’s performance.

```
    def act(self, observation):
        action, value, self._state, _ = self._model.step(
            transform_tuple(observation, lambda x: np.expand_dims(x, 0)),
            self._state,
            np.expand_dims(self._done, 0))
        # modified code
        if(self._restriction == True):
            # restrict action
            if self._psudoAPM > 100*self._alpha:
                action[0] = 0
            if action[0] == 0:
                self._psudoAPM = max(0, self._psudoAPM - 1)
            else:
                self._psudoAPM += 1
            # modified code end

        return action[0]
```

References

- Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., ... & Tsing, R. (2017). Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.
- https://github.com/deepmind/pysc2
- Lee, D., Tang, H., Zhang, J., Xu, H., Darrell, T., & Abbeel, P. (2018). Modular Architecture for StarCraft II with Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *14*(1), 187-193. https://doi.org/10.1609/aiide.v14i1.13033
- Liu, Y., Wang, W., Hu, Y., Hao, J., Chen, X., & Gao, Y. (2020, April). Multi-agent game abstraction via graph attention neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 34, No. 05, pp. 7211-7218).
- Sun, P., Sun, X., Han, L., Xiong, J., Wang, Q., Li, B., ... & Zhang, T. (2018). Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game. *arXiv preprint arXiv:1809.07193*. https://doi.org/10.48550/arXiv.1809.07193
- https://github.com/Tencent/PySC2TencentExtension
- https://github.com/Tencent/TStarBot1