# Appendix

## Mind-SC2

We design and implement a model that can be considered a mind-game which is called 'mind-SC2'. The model contains all the units and buildings of the three races. Every unit and building is implemented as a class. Unit and building attributes include information on their cost, health, armor and build time, all of which are the same as in the original StarCraft II. At the same time, the model approximates other basic features of StarCraft II, such as collecting resources, constructing buildings, producing units, etc. It is worth noting that we adopt a turn-based system similar to Go in the mind-game. Therefore, the time step of mind-game does not exactly correspond to the time step in StarCraft II which means they are different to a large extent. The design of mind-game can be seen in Figure 1.

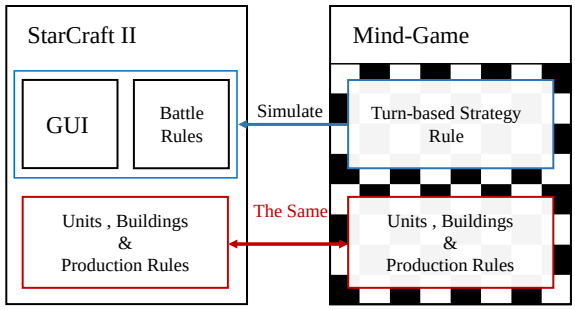



Figure 1: Design of Mind-Game.

In terms of the state transfer function, each action brings about a specific state transition, such as the production of a farmer's action. First, the algorithm determines whether the action satisfies the condition, such as whether the resource is satisfied, whether the technology is satisfied, and whether the population is satisfied. After that, if the conditions are met, the population is increased, and the resources consumed are subtracted. After that, the farmers' production enters the production queue of the building. When the farmers are produced out, the number of farmers increases by one. The code of mind-game is open source.

The most important and difficult part, the part of the battle, is designed like a classic turn-based tactical game (**?**). The basis of this design is a fact that the combat rules of SC2 are based on the war experiences of human own. The reason why human beings learn fast to play games is that humans do not learn from scratch. Before touching real-time strategy games, ordinary human players will have a general idea of the composition of a war and the key to win a war. The battle design of most games comes from real-world rules so that some of the battle parts of the game can be shared.

The complexity of SC2 comes from its graphical interface and the mechanism of real-time combat. Here, we replace this complex part with a tabular turn-based strategy game. The main factors that determine whether our agent will win are build orders and attack timing which are similar actions in both mind-game and source game. So when using module replacing, the agent's policy learned in mind-game can benefit training in source game. From this perspective, the principle of designing a simplified game is to approximate the complex modules in the original game with a simple module, while ensuring that the actions of the agent have a common semantic in both modules.

In the mind-game, agents are divided into two parties, the enemy and our side. An enemy can be either an agent that uses a policy or an agent that is implemented using a script. We set the enemy as a script agent. We adjust the speed at which the enemy increases the strength to control the difficulty of the mind-game. There are also several levels of difficulty in mind-game, from easy to hard.

## Training settings

The number of iterations of training is generally set to 800. Each iteration run 100 full-length game of SC2. One full-length game of SC2 is defined as an episode. The maximum number of frames for each game is set to 18,000. In order to speed up learning, we have adopted a distributed training method. We used 10 workers. Each worker has 5 threads. Each worker is assigned several CPU cores and one GPU. Each worker collects data on its own and stores it in its own replay buffer. Suppose we need 100 episodes of data per iteration, then each worker's thread needs to collect data for 2 episodes. Each worker collects the data of 10 episodes and then calculates the gradients, then passes the gradients to the parameter server. After the parameters are updated on the parameter server, the new parameters are passed back to each worker. Since the algorithm we use is PPO, the last old parameters are maintained on each worker to calculate the gradients.

Multi-processes parameters are as follows: the number of processes is set to 10, the number of threads in each process is set to 5, max-iteration $I$ is set to 500. It is noted that update-num $M$ is set to 500 on mind-game and 100 on the original environment because simulation speed in mind-game is much faster than the original environment.

PPO related parameters are as follows: $c2$ is set to $10^{-3}$ to encourage exploration, batch-size of PPO is set to 256, epoch-num is set to 10, initial learning rate is set to $10^{-4}$.

## Time Anlysis

The consumption of training time consists of several parts: 1. The time $t_\omega$ sampled in the environment, depending on the simulation speed of the environment. 2. The required sample size $m_\mu$ of the training algorithm. Due to the characteristics of the model-free reinforcement learning itself, a large number of samples are needed to learn a better policy. In addition, when using curriculum learning, the total sample size of the training is the sum of the sample sizes on all tasks. 3. The training time of the gradient descent algorithm, $t_\eta$. Therefore the total time overhead is $T_1 = (t_\omega + t_\eta) \cdot m_\mu$. Since the general sampling time is much longer than the gradient descent algorithm: $t_\omega \gg t_\eta$, then $T_1 \approx t_\omega \cdot m_\mu$.

Suppose our training process divides the task into steps of $k$. Step $k$ is our target task, and the previous step $k-1$ is the

pre-training process for curriculum learning. Therefore, we can write our training time as follows:

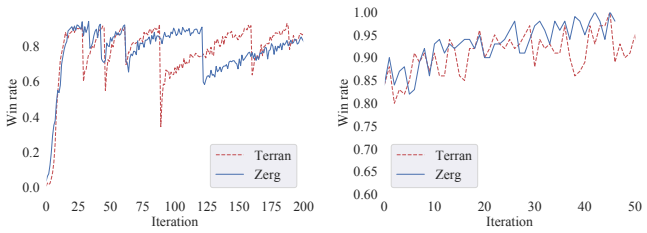
$$t_\omega \cdot m_\mu = t_\omega \cdot (m_{\mu_1} + m_{\mu_2} + \cdots + m_{\mu_k}) \quad (1)$$

In our approach, we moved the part of the curriculum to the mind-game. The simulation time in mind-game is much smaller than in the original environment. Assume that the time $t_\xi = 1/M \cdot t_\omega$ in the mind-game, and the amount of sample required by the last step of the task $m_{\mu_k}$ is $1/K$ of the total $m_\mu$. Then the time $T_2$ required by our algorithm can be denoted:

$$\begin{aligned} T_2 &= t_\xi \cdot (m_{\mu_1} + m_{\mu_2} + \cdots + m_{\mu_{k-1}}) + t_\omega \cdot m_{\mu_k} \\ &= 1/M \cdot t_\omega \cdot (K-1)/K \cdot m_\mu + t_\omega \cdot 1/K \cdot m_\mu \\ &= 1/M \cdot (K-1)/K \cdot t_\omega \cdot m_\mu + 1/K \cdot t_\omega \cdot m_\mu \\ &\approx 1/M \cdot 1 \cdot t_\omega \cdot m_\mu + 1/K \cdot t_\omega \cdot m_\mu \\ &= (1/M + 1/K) \cdot t_\omega \cdot m_\mu \\ &= (1/M + 1/K) \cdot T_1 \end{aligned} \quad (2)$$

This formula shows that the speedup ratio of the new algorithm is determined by the smaller value of the acceleration ratio $M$ and the last task sample ratio $K$. If we want to speed up the training time, the simulation speed needs to be as fast as possible and the most part of curriculum learning needs to be moved to the simulation part. Using these two methods at the same time can greatly improve the training speed.

## Test of other Races




(a) (b)

Figure 2: (a) The training process of Zerg agent and Terran agent in the mind-game. (b) The training process of Zerg agent and Terran agent in the source domian.

All previous methods were tested on only one race. Our method can efficiently train an effective agent, so we tested the results of training two other races, namely *Zerg* and *Terran*. The training process can be seen in 2.

Economic and combat settings of *Zerg* are similar to those of the *Protoss* above. The difference is that when *Zerg* builds a unit, it needs an extra resource – Larva. In mind-game, the number of Larvae is set to increase by one for every two steps. There will be an additional Larva for every two steps if any queen exists. In the source game, the number of Larvae is controlled by the game, i.e., the Hatchery will generate a Larva every 11 seconds. The Queen with Spawn Larva spell can inject 3 Larva eggs into the Hatchery and then the targeted Hatchery will generate 3 additional Larvae 29 seconds later. We set this spell to be automatically cast before each policy step ends. In addition, the *Zerg* Drones will disappear after building structures, which is also reflected in the mind-game.

## State space

The state space mainly includes two types, that is, source game features and mind-game features. The list of features of source game is shown in Table 1. The content of the mind-game features is based on Table 2.

## Macro actions

Here is how we generate macro actions. First we let the experts play 36 games including difficult level-1 to level3. In these games, we use the Protoss against the Terran, and only use the first two arms. After that, we saved the replays of the experts and analyzed these replays. The sequence of actions we dig through the PrefixSpan algorithm is as follows. The top 30 action sequences with the highest frequency of occurrence are listed in Table 3. It is worth noting that since all StarCraft 2 operations follow a form similar to English grammar, that is, the form of the subject plus verbs, the first action of all action sequences is necessarily the selection action. In addition, in the action sequence, many actions are smart screen operations. This operation produces different effects depending on the target of execution. Therefore, it is not helpful for building macro actions, and can be discarded. When we have dropped some duplicate action sequences, we can construct a collection of macro actions with the remaining ones. It should be noted that since macro actions are not code, we need to "translate" them into specific code. These codes need to be carefully written to ensure that they perform the sequence of operations in macro actions.

The current method is valid for the replays we use. Although it can be seen that the number of extracted macro actions is not large. But the benefit of being automated is that it can be extended to scenarios that use more arms and tactics. Although we have not made such an attempt in this paper, in the future, this method is worth a try. Interestingly, this idea doesn't just apply to StarCraft 2, it can be extended to any other environment using automated extraction operations, even to reduce the difficulty of solving problems in real-world scenarios.

Table 1: Source game state features

| features | remarks |
|---|---|
| opponent.difficulty | from 1 to 10 |
| observation.game-loop | game time in frames |
| observation.player-common.minerals | minerals |
| observation.player-common.vespene | gas |
| observation.score.score-details.spent-minerals | mineral cost |
| observation.score.score-details.spent-vespene | gas cost |
| player-common.food-cap | max population |
| player-common.food-used | used population |
| player-common.food-army | population of army |
| player-common.food-workers | population of workers(probes) |
| player-common.army-count | counts of army |
| player-common.food-army / max(player-common.food-workers, 1) | rate of army on workers |
| * num of probe, zealot, stalker, etc | multi-features |
| * num of pylon, assimilator, gateway, cyber, etc | multi-features |
| * cost of pylon, assimilator, gateway, cyber, etc | multi-features |
| * num of probe for mineral and the ideal num | multi-features |
| * num of probe for gas and the ideal num | multi-features |
| * num of the training probe, zealot, stalker | multi-features |

Table 2: Mind-game state features

| features | remarks |
|---|---|
| observation.player-common.minerals | minerals |
| observation.player-common.vespene | gas |
| observation.score.score-details.spent-minerals | mineral cost |
| observation.score.score-details.spent-vespene | gas cost |
| minerals + mineral cost | mineral collected |
| gas + gas cost | gas collected |
| player-common.food-cap | max population |
| player-common.food-used | used population |
| player-common.army-count | counts of army |
| * num of probe, zealot, stalker, etc | multi-features |
| * num of pylon, assimilator, gateway, cyber, etc | multi-features |
| * num of probe for gas and mineral | multi-features |

Table 3: Prefixspan-Result (30 most frequently occurring)

| action sequences | frequency |
|---|---|
| select-point(unit name: Protoss.Probe) → Harvest-Gather-Probe-screen | 2711 |
| select-point(unit name: Protoss.Probe) → Smart-screen | 2253 |
| select-point(unit name: Protoss.Nexus) → Train-Probe-quick | 1421 |
| select-point(unit name: Protoss.Probe) → Build-Pylon-screen | 1298 |
| select-point(unit name: Protoss.Nexus) → Smart-screen | 1030 |
| select-point(unit name: Protoss.Nexus) → select-army | 968 |
| select-point(unit name: Protoss.Probe) → Build-Gateway-screen | 814 |
| select-point(unit name: Protoss.Gateway) → Train-Zealot-quick | 762 |
| select-point(unit name: Protoss.Probe) → Build-Pylon-screen → Harvest-Gather-Probe-screen | 659 |
| select-point(unit name: Protoss.Gateway) → Train-Stalker-quick | 621 |
| select-point(unit name: Protoss.Nexus) → select-army → Attack-Attack-screen | 581 |
| select-point(unit name: Protoss.Probe) → select-army | 574 |
| select-point(unit name: Protoss.Nexus) → Train-Zealot-quick | 536 |
| select-point(unit name: Protoss.Gateway) → select-army | 467 |
| select-point(unit name: Protoss.Probe) → Build-Gateway-screen → Harvest-Gather-Probe-screen | 466 |
| select-point(unit name: Protoss.Nexus) → Train-Stalker-quick | 325 |
| select-point(unit name: Protoss.Gateway) → Smart-screen | 300 |
| select-point(unit name: Protoss.Gateway) → select-army → Attack-Attack-screen | 284 |
| select-point(unit name: Protoss.Probe) → Attack-Attack-screen | 260 |
| select-point(unit name: Protoss.Probe) → Build-Pylon-screen → Smart-screen | 258 |
| select-point(unit name: Protoss.Probe) → select-army → Attack-Attack-screen | 254 |
| select-point(unit name: Protoss.Probe) → Build-Assimilator-screen | 253 |
| select-point(unit name: Protoss.Probe) → Train-Zealot-quick | 243 |
| select-point(unit name: Protoss.Gateway) → Cancel-BuildInProgress-quick | 209 |
| select-point(unit name: Protoss.Probe) → Smart-screen → Build-Pylon-screen | 194 |
| select-point(unit name: Protoss.Probe) → Build-CyberneticsCore-screen | 182 |
| select-point(unit name: Protoss.Probe) → Smart-screen → select-army | 166 |
| select-point(unit name: Protoss.Probe) → Smart-screen → Harvest-Gather-Probe-screen | 164 |
| select-point(unit name: Protoss.Probe) → Smart-screen → Build-Gateway-screen | 147 |
| select-point(unit name: Protoss.Probe) → Harvest-Gather-Probe-screen → select-army | 147 |