A Model-Based Method for Solving an MDP Scenario in StarCraft II

By Gengyu Zhang & Jingyi Li

Background & Related Work (Background)

- StarCraft II

StarCraft II, created by Blizzard Entertainment, is the most successful real-time strategy (RTS) of all time. There are various game modes in the game, where the most common one is the traditional 1v1 game. This game mode requires a player to carefully balance big-picture management of their economy – known as macro – along with low-level control of their individual units – known as micro. The challenges in the research of mastering SC2 requires breakthroughs in the following main aspects: game theory, imperfect information, long-term planning, real-time and large action space.

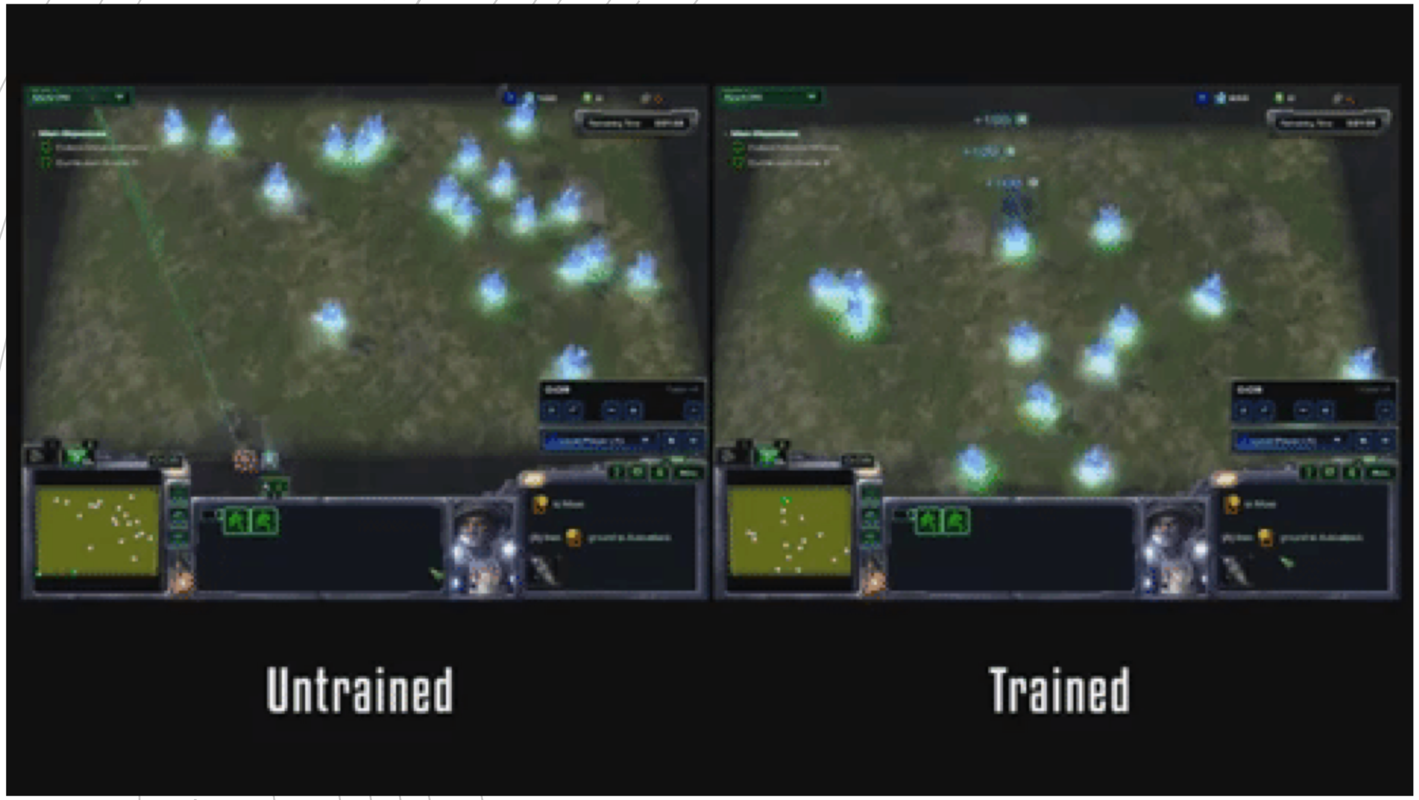
- PySC2

PySC2 is DeepMind's Python component of the SC2 Learning Environment (SC2LE). It exposes Blizzard's SC2 Machine Learning API as a Python RL Environment. PySC2 provides an interface for game agents to interact with SC2, getting observations and sending actions. Especially, PySC2 provides a series of mini-game maps, which specifically work for optimizing decision making of an agent with respect to its micro – controlling individual units to accomplish some designed tasks. Our work is based on one of these mini-games.

Background & Related Work (Related Work)

- Now the state-of-the-art work in training an agent to play SC2 belongs to DeepMind. Different from our work, nevertheless, they applied a DRL approach to observe the game environment and to select appropriate actions. Their SC2 agent is known as *AlphaStar*, whose behavior is generated by a deep neural network that receives input data from the raw game interface (a list of units and their properties), and outputs a sequence of instructions that constitute an action within the game. The neural network architecture applies a transformer torso to the units, combined with a deep LSTM core, an auto-regressive policy head with a pointer network, and a centralized value baseline.

Proposed Problem



GIF of Untrained agent VS Trained agent demo
(Play the current page to watch)

We are going to select a scenario from StarCraft II and try modeling it as an MDP to optimize decision-making problem in it. In details, the scenario consists of two units (marines) controlled by the agent and a bunch of neutral mineral shards scattering in the map randomly. The task is to control the two units to collect as many those shards as possible within a given time, where a unit collects a mineral shard by moving to the location of the shard. Thus, an optimal trajectory for the units is expected to be found, and the cooperation of the two units should also be considered.

Approach (1/3)

Inputs:

- *S*: A set of states. A state is represented as a tuple of location coordinates of remaining mineral shards and location coordinates of the two units.
- *A*: A set of actions which are available in given states. Generally, the available actions include: no operation, selecting units, moving units to a location in the map, stopping the units and holding position.
- *γ*: The discounted factor.
- *c*: The prior count.

Compute and Update:

- *V (S, A)*: Array of expected utilities.
- *R (S, A, S)*: Array representing the reward function. *R* (*s*, *a*, *s'*) represents that in the state *s*, the agent carries out the action a and it turns into the state *s'*.
- *T (S, A, S)*: Array representing the model of dynamics. *T* (*s*, *a*, *s'*) is the count of the number of times that the agent carries out a in state *s* and ends up in *s'*. The counts are added to the prior count, as in a Dirichlet distribution, to compute the probabilities of the transition model.
- *P (S, A, S)*: The transition function. *P* (*s*, *a*, *s'*) represents the probability that the agent carries out *a* in *s* and results in *s'*.

Approach (2/3)

- **The Model-based Algorithm:**

```
controller ModelBasedController (S, A, γ, c)
    inputs: S, a set of states; A, a set of actions; γ, the discounted factor; c: prior count.
    local Variables: V (S, A), double array; R (S, A, S), double array; T (S, A, S), int array.
    initialize V (S, A) arbitrarily; initialize R (S, A, S) arbitrarily; initialize T (S, A, S) to zero; observe
    current state s; select and carry out action a.
    repeat forever:
        observe reward r and state s'
        select and carry out action a
        $T(s,a,s') \leftarrow T(s,a,s') + 1$
        $R(s,a,s') \leftarrow R(s,a,s') + (r - R(s,a,s'))/T(s,a,s')$
        $s \leftarrow s'$
        repeat
            select state $s_1$, action $a_1$
            let $P = \sum_{s_2}(T(s_1,a_1,s_2) + c)$
            $V(s_1,a_1) \leftarrow \sum_{s_2}(T(s_1,a_1,s_2) + c)\,/(P)(R(s_1,a_1,s_2) + \gamma max_{a_2} V(s_2,a_2))$
        Until an observation arrives
```

Approach (3/3)

- **General Narrative of the Algorithm**
  - In the game environment, we do not own the prior knowledge of the transition model as that in the familiar grid world we learned about in class, so we need to learn the transition model based on experience when the game is running.
  - The algorithm keeps track of the expected utilities for all states $V(S, A)$ in each step, but it also maintains a model of dynamics, represented here as T, where $T(s, a, s')$ is the count of the number of times that an agent conducts a in the state s and ends up in s'. The counts are then added up to the initialized and uniformly distributed prior count c to compute the probabilities of the transition model. This algorithm assumes a common prior count for all transitions. The $R(s, a, s')$ array maintains the averaged reward for all transitions from s to s' by acting a. After each action, the agent observes the reward r and the resulting state s'. It then updates the transition-count matrix T and the average reward R. It then conducts a value iteration, using the updated probability model derived from T and the updated reward model. Finally, when the temporal difference is less than a given small value, the iteration terminates.

Demonstration

- We plan to compare the performance of our model-based agent with that of two PySC2 built-in agents (a random agent and a scripted agent), based on two respective criterion – the **number of mineral shards collected** within the same given time and **time consumed** for collecting all mineral shard in the map. A more intuitive comparison of replays broadcasting the actual behaviors of the three agents will also be presented.

Deliverables

- Code - The code will be written in Python. The project should contain two classes: one for generalizing states, actions and reward function from APIs provided by the game, introducing the discounted factor and initializing the transition model to construct the MDP model; and the other is for implementing the value iteration and update the transition model.

Responsibilities

- Gengyu Zhang is responsible to implement the model-based algorithm and to compare our agent with another two built-in agents and evaluate the performance.
- Jingyi Li is responsible to generalize the necessary components from the game APIs and to construct the MDP model.