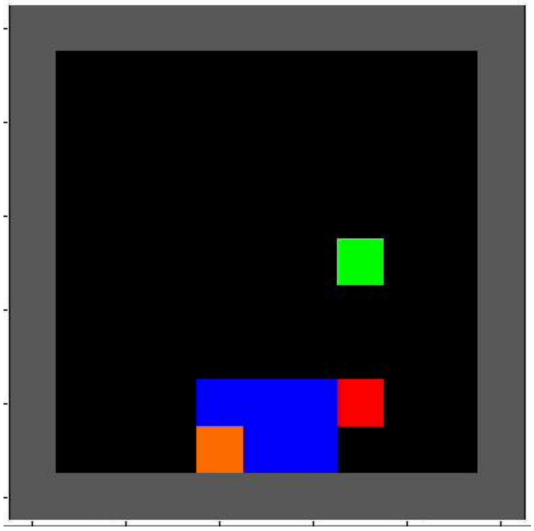
Project3

1. IQL (Independent Q-learning)

The **Food Collector** is implemented as a grid world, with a 11 x 11 grid. Agents are colored red and orange for better identification, the food is colored green, home is colored blue and walls are colored grey. The environment is based on and fully compatible with OpenAI Gym API.



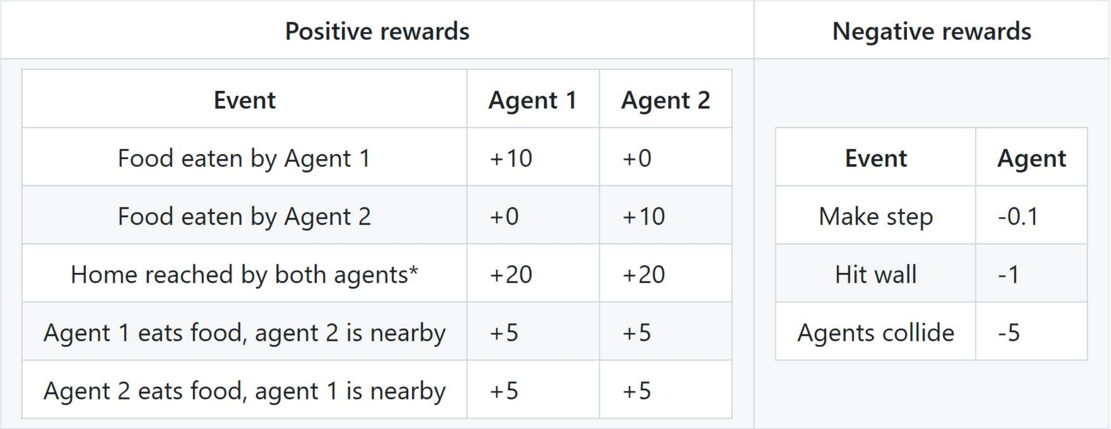
The objective of the game is simple: one of the agents needs to eat the food and then they both need to return home. The game only ends if the food is eaten and both agents are in the home area. Moreover, the agents get bonus points if they are both next to the food when it is eaten. Therefore, the expected optimal behaviour is: (1)Both agents get close to the food (agent that spawns closer to food waits for the other agent); (2)One of them eats the food; (3)They both return home straight after. On the contrary, a greedy behaviour would be for the agent closer to the food to immediately eat it.

For simplicity and faster training, we use a feature vector for the state representation. The observable state space for each agent consists of a 7-element vector:

- Two elements to describe the relative x and y distance to the food
- Two elements to describe the relative x and y distance to the home
- Two elements to describe the relative x and y distance to the other agent
- A binary element describing whether food has been eaten yet or not

Each agent has 4 actions to chose: move up, down, left or right. If an illegal move is chosen, such as moving into a wall or colliding with another agent, the agents stay in place.

The reward table is as follows:



Positive rewards		
Event	Agent 1	Agent 2
Food eaten by Agent 1	+10	+0
Food eaten by Agent 2	+0	+10
Home reached by both agents*	+20	+20
Agent 1 eats food, agent 2 is nearby	+5	+5
Agent 2 eats food, agent 1 is nearby	+5	+5

Negative rewards	
Event	Agent
Make step	-0.1
Hit wall	-1
Agents collide	-5

*Reward given and game ends only if the food is eaten

IQL is a simple method for multi-agent reinforcement learning, whose core idea is to decompose the multi-agent problem into multiple independent single agent learning tasks, that is, to give each agent a separate Q-learning algorithm to perform. It avoids joint action space computation and has low computational complexity; However, due to the shared environment and the fact that the environment changes with the policies and states of each agent, the environment is dynamically unstable for each agent, so this algorithm cannot guarantee convergence.

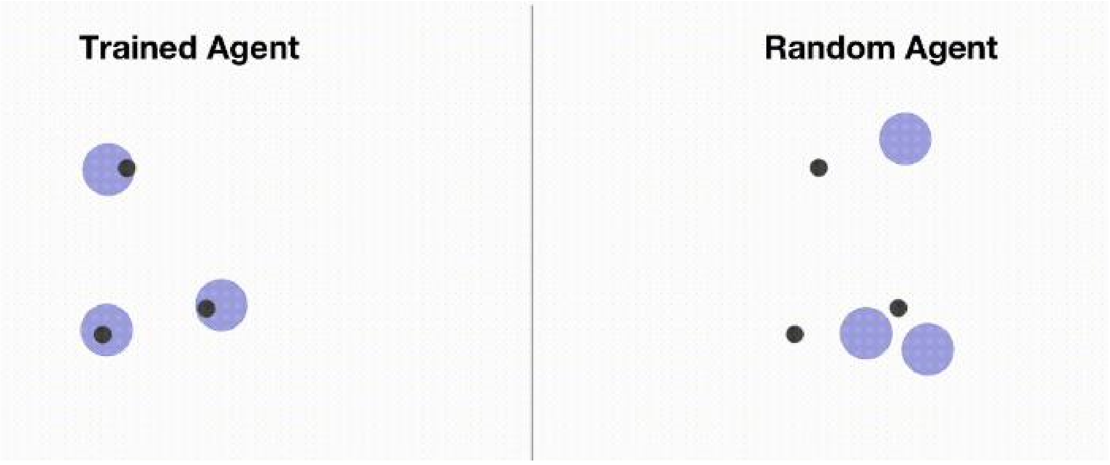
Q1: Implement the IQL algorithm on the Food Collector environment in IQL.*ipynb*

In this project, we use raw q-learning instead of DQN. Your main task is to complete updating Q-table and taking action.

2. MAPPO (multi-agent variant of PPO)

Simple Spread is one of the popular environments in MPE(Multi Particle Environments). This environment has 3 agents and 3 landmarks. At a high level, agents must learn to cover all the landmarks while avoiding collisions. More specifically, all agents are globally rewarded based on how far the closest agent is to each landmark (sum of the minimum distances). Locally, the agents are penalized if they collide with other agents (-1 for each collision).

For more details, you can search: https://pettingzoo.farama.org/environments/mpe/simple_spread/.



MAPPO algorithm is a multi-agent variant of PPO. In the last assignment, you implemented the PPO algorithm. In this one, you will read and understand the overall framework of multi-agent code and fill in the blanks to complete the core parts of MAPPO, including **calculating loss** and **parameter updating**. The Actor and Critic network and the training parameters are all fixed, you don't need to modify them.

Q2: Implement the MAPPO on the Simple-Spread environment in r_mappo.py

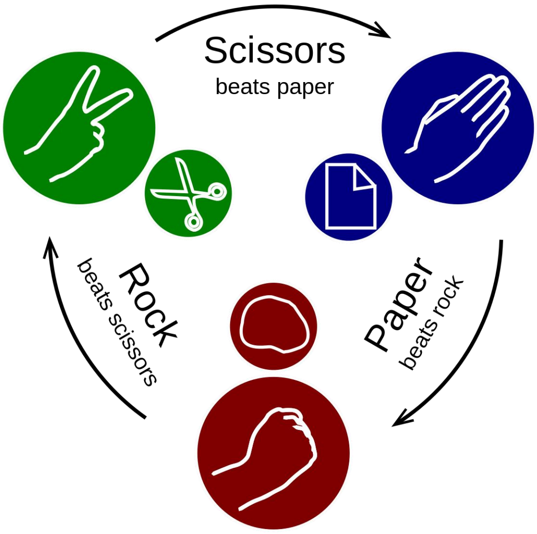
You can start with train.py to follow the code framework, the environment-specific code is in env_core.py. You need to complete the two functions in r_mappo.py, cal_value_loss() and ppo_update(), the related passed parameters and return required return value are fixed, please do not modify yourself.

3. PSRO(Policy-Space Response Oracles)

The env used is **Rock-Paper-Scissors** built on gym API.

This is a two person finite zero sum game. Win +1 point, lose -1 point, draw 0 point.

The observation space is Discrete(3): (ongoing 0 / lose 1 / win 2). Note that when win/lose happens, an episode would be ended. So the state player see is always "ongoing 0". This means that in train.py, for the matrix of policy "pi", you only need to focus on the first row, which is the probability of the agent taking each action in the game state of "ongoing 0".



In InRL (Independent Reinforcement Learning), each agent treats [other agents+real environment] as the "environment", which leads to the problem of non-stationary environment. Non-stationarity refers to the fact that the state transition function of the environment is constantly changing. Directly applying the method of single agent reinforcement learning to multi-agent reinforcement learning will face the problem of non-stationary environment.

The motivation behind the PSRO is the flaw of InRL. PSRO is a natural generalization of Double Oracle where the meta-game's choices are policies rather than actions. Unlike previous work, any meta-solver can be plugged in to compute a new meta-strategy(In this project, linear programming is used to solve Nash equilibrium).

Q3: Implement the PSRO algorithm on Rock-Paper-Scissors game in *psro.py*

The RL algorithm(Q-learning) used to obtain the best response has been implemented in agent.py, please do not modify and use directly. Your main task is to obtain the payoff matrix, and use Q-learning to obtain the best response.