

# HW7 Discover Reinforcement Learning in the Game Snakes and Apples

Author: Shiyu Chen

Date: May 31, 2023

## Introduction & Overview

In this project, we are going to explore reinforcement learning (RL) by playing the Snake and Apples game. The game is played on a 4x4 grid. The snake moves continuously, and eating apples increases the snake's length. The game ends if the snake collides with the edge or its body. The objective is to earn points by eating apples and achieving a high score. The algorithm that is used to play the game is driven by the reward function. Our goal is to investigate the game plan by shaping the parameters in the reward function.

## Theoretical Background

This game can be structured as Markov Decision Processes (MDPs). In an MDP, an *agent*, the decision maker, interacts with the *environment* it is placed in. This interaction happens over time, with the agent receiving information about the environment's *state* at each time step. Based on this information, the agent chooses an *action* to take. The environment then transitions to a new state, and the agent receives a *reward* as a result of its previous action.

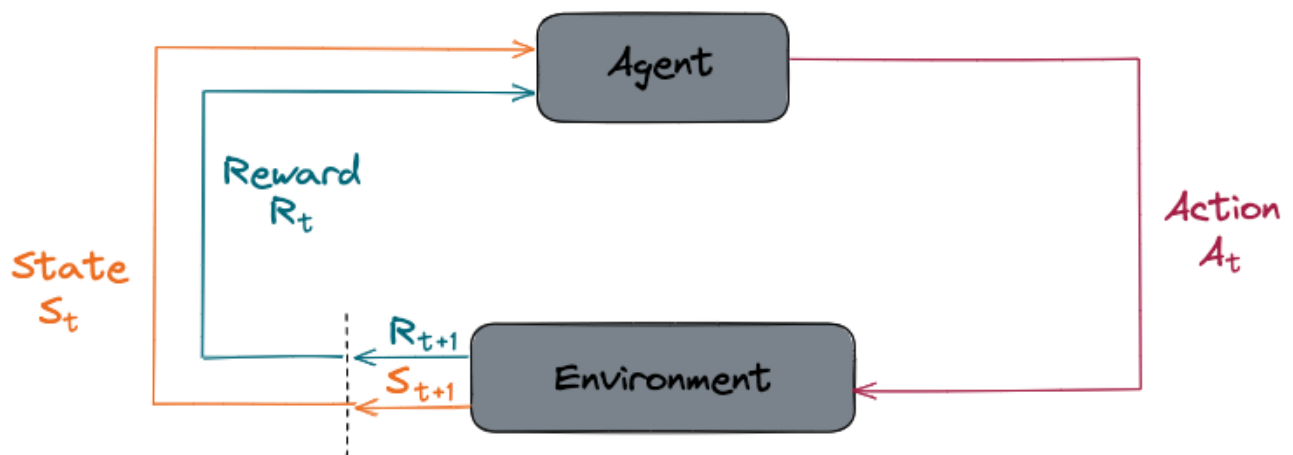


Figure 1: [1] Diagram for MDP

The code that is used to play the game is based on the Q-learning technique in reinforcement learning.

*Reinforcement learning* (RL) focuses on how an agent can learn to make optimal decisions in an environment to maximize its overall reward. It involves the interaction between an agent and an environment, where the agent learns from the feedback it receives in the form of rewards or penalties. The criteria for measuring the rewards is expected discounted return:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Where  $t$  represents each time step,  $\gamma$  is the discount rate, and  $G_t$  is the cumulation of discounted rewards received at each time step. Therefore, the agent's goal is to maximize the expected discounted return of rewards. To our snake, it goes in the direction that has the largest probability to eat as many apples without hitting the edge or its own body.

In each state (each movement), the snake has four options to action, either going up, down, left or right. We use policy ( $\pi$ ) to figure out how possible is it for the snake to go in any direction (action) based on its current states like its position, length and the position of apple. The expression  $\pi(a|s)$  means the probability of taking action  $a$  in state  $s$  under the policy  $\pi$ .

To define how good the action is for agent in state  $s$  under policy  $\pi$ , we use Action-value function:

$$q_{\pi}(s, a) = E_{\pi}[G_t | S_t = s, A_t = a]$$

This function is also referred to as the Q-function, and its result, Q-value, will be used in Q-learning technique.

The goal of Q-learning is to find the optimal policy by learning the optimal Q-values for each state action pair, and it iteratively updates Q-values for each state action pair by Bellman equation

$$q_*(s, a) = E[R_{t+1} + \gamma \max_a q_*(s', a')]$$

Until the  $q(s, a)$  converges to its optimal function  $q_*$ .

The Q-learning store the Q-value in Q-table, and in each iteration update the Q-table by equation:

$$q^{new}(s, a) = (1 - \alpha) \underbrace{q(s, a)}_{\text{old value}} + \alpha \overbrace{\left( R_{t+1} + \gamma \max_a q(s', a') \right)}^{\text{new value}}$$

Where  $\alpha$  represent the learning rate, which determines how much information we want to keep from the old value when we are calculating the new Q-value.

In each episode, the snake can choose whether to follow the route it walked before and earn the same reward as before, or try a new route which has the possibility to earn more rewards, and also possibility to fail. This is called exploration vx. Exploitation dilemma.

To address this problem, we define an exploration rate  $\epsilon \in [0, 1]$  to decide whether to explore or exploit. With  $\epsilon=1$ , it is 100% certain that the snake will explore the environment. As the snake learn more about the environment, the  $\epsilon$  should decay and therefore the program becomes more "greedy" and less possible to explore unknown things.

## Game Exploration and Results

The condition of success is to get 24 scores, and snake's body will fill all 16 grids. Eating an apple add 1 score, and the body of the snake extends to 1 more block long. The last apple has 10 score, so the snake needs to take 15 apples to succeed.

Our task is tuning the parameters:

**N\_episodes:** the episode of training. If the n\_episodes is too small, there will be no enough data to refine the Q-table. More episodes means more training and will earn higher score (Figure 3), but also take longer time to process.

**Epsilon:** the exploration rate. Larger the epsilon means the snake will be more possible to explore the environment.

**Gamma:** the discount rate, which means how far should the snake to think about the future rewards. Larger gamma usually takes longer time to process.

**Rewards:** includes Losing move, inefficient move, efficient move and winning move.

**Losing move** should be a negative number, which means any move that leads to lose should not be taken. If the losing move is too small, the snake will go back and force in several blocks as it is too "afraid" to lose.

**Inefficient move** means the move is not helpful for getting the apple but might be helpful for avoiding losing the game.

**Efficient move** is the move that aims to get the apple. The efficient and inefficient move should keep a balance or the snake will also fall into an infinite loop.

**Winning move** is the move that will eat apples. It should be a large number so the snake has "motivation" to eat the apple. (figure 4)

As there are only 16 grids on the playboard, the snake must fill every block to win the game. Therefore, when the snake approaches the apple, it might not take the shortest path, but in a roundabout way that squeezes its body into smaller spaces to better fill the grid.





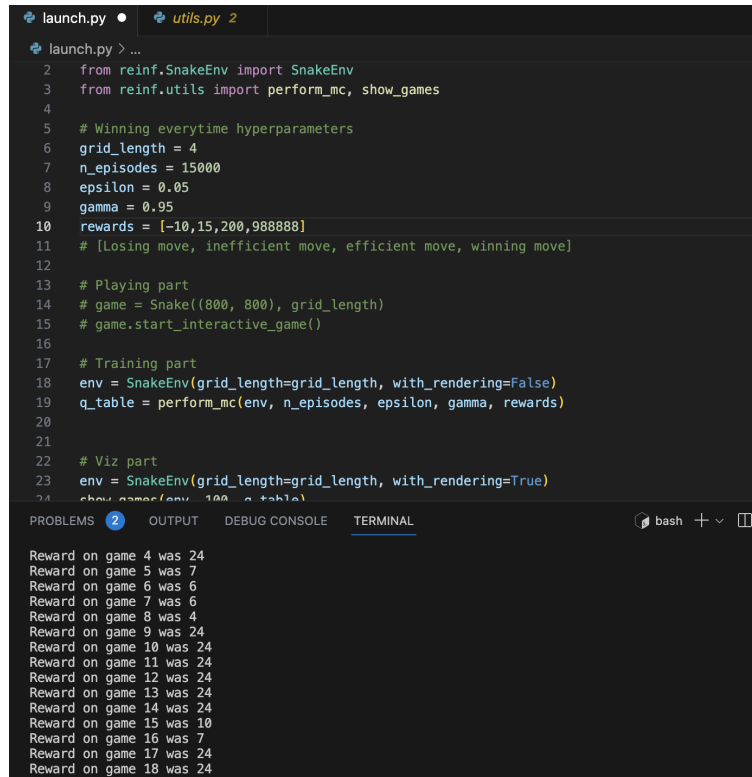
```
4
5 # Winning everytime hyperparameters
6 grid_length = 4
7 n_episodes = 15000
8 epsilon = 0.05
9 gamma = 0.95
10 rewards = [-10,10,200,988]
11 # [Losing move, inefficient move, efficient move, winning move]
12
13 # Playing part
14 # game = Snake((800, 800), grid_length)
15 # game.start_interactive_game()
16
17 # Training part
18 env = SnakeEnv(grid_length=grid_length, with_rendering=False)
19 q_table = perform_mc(env, n_episodes, epsilon, gamma, rewards)
20
21
22 # Viz part
23 env = SnakeEnv(grid_length=grid_length, with_rendering=True)
24 show_games(env, 100, q_table)
25
```

100% | 15000/15000 [00:18<00:00, 799.19it/

Reward on game 0 was 4  
Reward on game 1 was 24  
Reward on game 2 was 9  
Reward on game 3 was 24  
Reward on game 4 was 5  
Reward on game 5 was 24  
Reward on game 6 was 8  
Reward on game 7 was 7  
Reward on game 8 was 24  
Reward on game 9 was 24  
Reward on game 10 was 6  
Reward on game 11 was 8  
Reward on game 12 was 7  
Reward on game 13 was 9

Figure 5. Result with epsilon = 0.05

As what we found in previous attempts, higher winning move may raise the score of game, I raise the winning move from 988 to 988888. The result is consistent with our expectation, the rate of success raise to 8/10. (Figure 6)



```
launch.py • utils.py 2
launch.py > ...
2 from reinf.SnakeEnv import SnakeEnv
3 from reinf.utils import perform_mc, show_games
4
5 # Winning everytime hyperparameters
6 grid_length = 4
7 n_episodes = 15000
8 epsilon = 0.05
9 gamma = 0.95
10 rewards = [-10,15,200,988888]
11 # [Losing move, inefficient move, efficient move, winning move]
12
13 # Playing part
14 # game = Snake((800, 800), grid_length)
15 # game.start_interactive_game()
16
17 # Training part
18 env = SnakeEnv(grid_length=grid_length, with_rendering=False)
19 q_table = perform_mc(env, n_episodes, epsilon, gamma, rewards)
20
21
22 # Viz part
23 env = SnakeEnv(grid_length=grid_length, with_rendering=True)
24 show_games(env, 100, q_table)
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

```
bash + v □
Reward on game 4 was 24
Reward on game 5 was 7
Reward on game 6 was 6
Reward on game 7 was 6
Reward on game 8 was 4
Reward on game 9 was 24
Reward on game 10 was 24
Reward on game 11 was 24
Reward on game 12 was 24
Reward on game 13 was 24
Reward on game 14 was 24
Reward on game 15 was 10
Reward on game 16 was 7
Reward on game 17 was 24
Reward on game 18 was 24
```

Figure 6. Result with large winning move

## Conclusion

In conclusion, the key of making program succeed is to have a appropriate ratio of four rewards, and small epsilon and large winning move raise the opportunity to win.

This project provides an opportunity to get familiar with reinforcement learning. By exploring the parameters in algorithm, we better understand how the game plan structured.

## Citation:

[1] DeepLizard, "Keras Tutorial: How to Get Started with Keras, Deep Learning, and Python," in DeepLizard, 2018. [Online]. Available: <https://deeplizard.com/learn/video/my207WNoeyA>. [Accessed: May 23, 2023].