Snake-Agent

Project Overview

The project uses the snake game as the foundation, and the goal is to achieve Q-learning reinforcement learning, so that the agent's performance in the game gradually conforms to the rules and the score improves.

Github Link: s1eeping-king/Snake-Agent

Youtube Link: Snake-Agent

1. Game Design

1.1 Objective and Rules

The goal of the game is to constantly optimize your strategy and get as high as possible in your score. The rules are as follows:

1. Movement Control: The snake chooses the direction of going straight, left or right to move.

2. Food: Eating food will increase the score and the length of the snake.

3. Collision Detection: The game ends when the snake's head touches the border or itself.

4. Scoring system: Score points for eating food, deduct points for hitting walls or bodies, score points for getting close to food, and display the total score at the end of the game.

1.2 State and Action Space

Define the state space of the game (all state records of the Agent).

Variables and their corresponding explanations:

danger_straight, danger_right, danger_left, *# Danger in front, right, and left directions*

dir_left, dir_right, dir_up, dir_down, *# Current movement direction*

food_left, food_right, food_up, food_down *# Relative food direction*

Define the game's action space (all actions that agents can perform):

action_0, action_1, action_2 #Go straight，Turn left，Turn left

1.3 Reward Design

In this game, the rewards are designed to guide agents to learn how to survive in the game and achieve higher scores. Because there aren't many reward categories required for this game, there is no specific reward function. The settings for positive and negative rewards are as follows:

Positive Rewards:

- **Eat food:** When the snake's head coincides with the food, the intelligence will receive a positive reward of +10. This reward encourages the agent to seek out and eat food, thereby increasing the score.

Negative Rewards:

- **Collision:** If the position of the snake's head overlaps a wall or itself, the intelligence will collide and receive a negative reward of -10. This design is designed to punish improper movement and encourage the agent to avoid collisions.
- **The Manhattan distance between snakeheads and food:** In the case of no food, the

agent will receive a small negative reward based on the Manhattan distance to the food. The calculation is as follows: -food_distance / (self.width + self.height). This design allows the agent to move in the direction of the food, taking into account the distance from the food when moving.

Through the design of positive and negative rewards, the agent learns:

- Eating food is a beneficial behavior, so it will be preferred to move in the direction of the food.
- Collision avoidance is key to survival, so learn how to avoid walls and yourself.

1.4 Game Design Documentation

Traditional Snake Game: The traditional snake works on a simple principle, it is to choose an action, if it hits a body or a wall, the game is over and the score is recorded, and the goal of the game is to maximize the amount of food eaten to earn high points. Figure 1.1

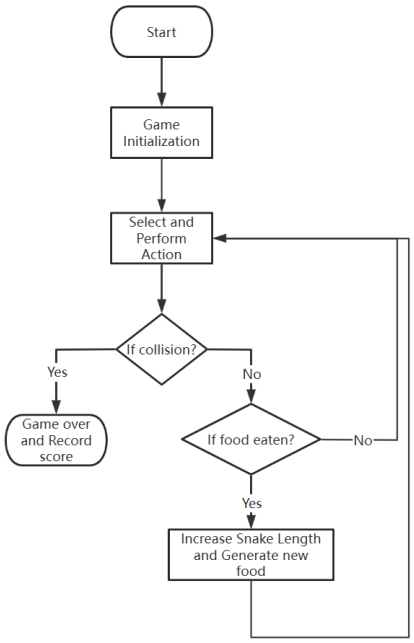

Figure 1.1 Flowchart of a traditional snake game

Agent Training Process: Figure 1.2 shows the agent training process of this game. Game training begins, first initialize the Q table, and then initialize the game environment. Then, determine whether the current state vector exists, and if not, enter the process of creating a state vector (Figure 1.3). If it exists, the current state vector is directly obtained. Select an action based on the Q table and exploration rate, and then execute the selected action. Then enter the reward calculation link (Figure 1.4), and update the current state according to the reward calculated by the reward. Then determine if there is a collision and whether the maximum number of training sessions has been reached:

If there is no collision: The snake moves, and the snake again determines whether the current state exists, entering the loop.

In the event of a collision: Determine whether the maximum number of training sessions has been reached:

If the maximum number of training sessions is not reached: Re-initialize the

game environment, but the Q-table will not be initialized and will enter the loop.

If the maximum number of training sessions is reached: The game is over, save the Q sheet

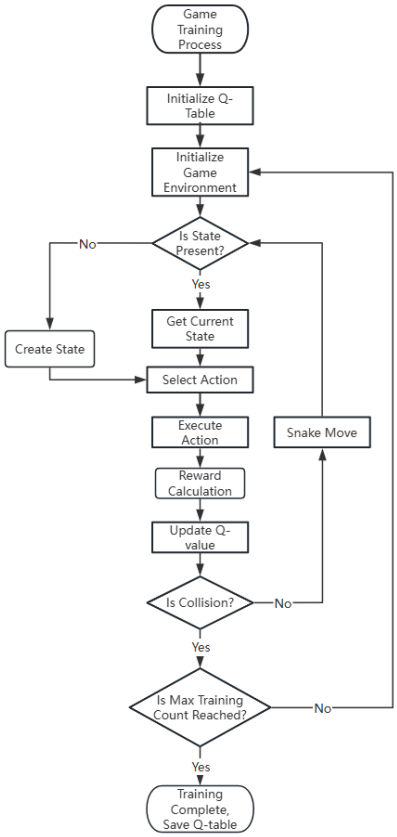

Figure 1.2 Agent Training Flowchart

State Creation Process: The purpose of state creation is to solve the problem that it takes time to initialize the state and to speed up the operation efficiency, as shown in Figure 1.3. Get the current head position, then calculate the food orientation, whether the three directions (except backwards) are dangerous, and the current direction of movement, and finally combine and create a state vector.

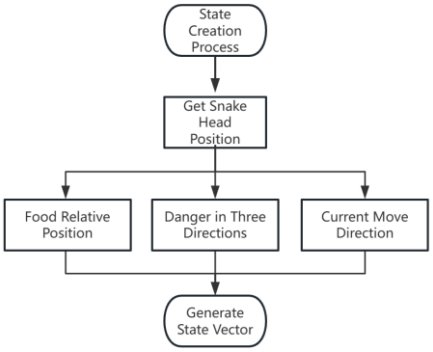

Figure 1.3 State Creation Process

Reward Calculation Process: The reward calculation can also be called Q value update, which determines the behavior result of this step to calculate its score, as shown in Figure 1.4. Enter the reward calculation process to determine whether there is a collision:

If Yes: the penalty is returned;

If No: Determine if the food has been eaten:

If Yes: return rewards;

If No: Penalties are returned based on distance from food

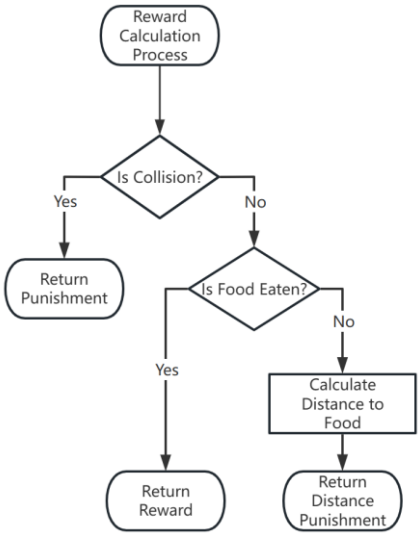

Figure 1.4 Reward Calculation Process

2. Q-Learning Implementation

2.1 Algorithm Implementation

The Q-Learning algorithm is written in Python in this project, and the implementation can be divided into several key parts: state space design (1.2), action space design (1.2), training process (1.4), Q table structure, key parameter setting, and core algorithm implementation. The first three parts have been detailed in the previous chapters, and the remaining three are introduced below

Q table structure: The Q table of this project is stored using a dictionary, the key is the state tuple, and the value is an array containing 3 action values. This Q table has a dynamic expansion function, and when a new state is encountered, the corresponding Q value is automatically initialized as a zero vector (refer to Figure 1.3).

Key parameter settings: Q-learning for this project has the following five key parameters:

- Learning rate (alpha) = 0.1: Control the rate at which new knowledge is learned
- Discount Factor (GAMMA) = 0.9: Balance immediate and future rewards
- Initial Exploration Rate (EPSILON) = 1.0: Completely random exploration at the beginning
- Minimum exploration rate (epsilon_min) = 0.01: Guaranteed minimum exploration probability
- Exploration Rate Decay (epsilon_decay) = 0.995: Controls the speed at which exploration transitions to exploitation

Core algorithm implementation: The core algorithm can be divided into two parts:

- Based on ε- Greedy strategy action selection:
  - Randomly select actions with a probability of ε (Explore)
  - Choose the action with the highest Q value with a probability of 1-ε (utilize)
  - The exploration rate decays with training
- Q value update:
  Core update formulas using Q-Learning:
  $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R + \gamma \cdot \max Q(s',a')]$
  Thereinto:
  - s,a is the current state and action
  - R is an instant reward
  - s' is the next state
  - max Q(s',a') is the maximum Q value for the next state
  - α is the learning rate
  - γ is the discount factor

2.2 Exploration and Exploitation

The learning strategies for this project have had a significant effect. As shown in Figure 2.1, this graph records the score for each game and the average score for each 100 rounds.

As you can see from the graph, although the score fluctuates greatly (most likely due to the epsilon-greedy exploration strategy), the average score per 100 rounds gradually rises until it converges to a maximum of around 400 rounds, which is around 25 points.

Figure 2.1 Score Records

2.3 Learning Policy

As mentioned in 2.2, our training results are good, and the score is always going up. epsilon-greedy exploration: determine whether the exploration rate is greater than a random number of 0~1, if it is greater, it will be randomly selected; If it is less than that, select the highest Q value in the Q table to take the next action. As shown in Figure 2.2:

```
# ε-贪婪策略选择动作
if random.random() < self.epsilon:  # 探索
    return random.randint(0, self.action_size - 1)
return np.argmax(self.q_table[state_key])  # 利用
```

Figure 2.2 Epsilon-greedy Code

Learning rate: As you can see from the figure, the learning rate is about 70 rounds to the lowest, and from the performance of Figure 2.1, it can be seen that the snake is learning correctly.

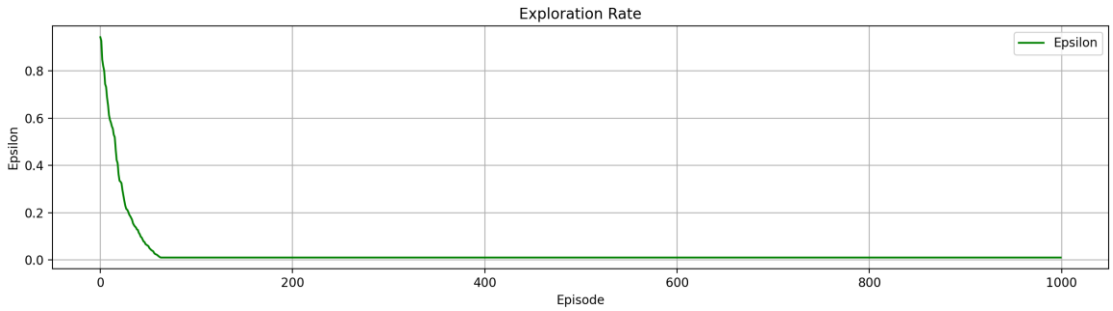

Figure 2.3 Learning Rate

Discount Factor: The setting of the discount factor is related to Reward, and the values of Reward and Gamma for this item have been tested to be appropriate.

2.4 Q-learning implementation

Instructions for the implementation of Q-learning have been introduced in 2.1, 2.2, 2.3

3. Game Interaction

3.1 Interaction Mechanism

The interaction of the agent is mainly based on the action space of the Q table and random actions. Contextual awareness is also achieved through the state space of Q tables.

During training, the agent selects actions through the ε-greedy strategy through the get_action (*self*, *state*) method, executes the action through the step(action) method, and calls the get_state(). The method returns a new state, and then updates the Q value according to the returned state, and then continues to select the action to enter the loop. This training process enables the agent to learn how to interact with the environment and improve the performance of the interaction.

3.2 UI Introduction

The UI design of this project is relatively simple, with Score at the top as the score and Episode at the bottom as the number of training sessions. The snake's head is made up of white squares and blue squares, with a green body and red squares for food.

The current status of the agent is displayed in the form of the head of the snake.

The display of the reward is an increase in the Score

The penalty is displayed as the addition of Episodes, the zeroing of scores, and the initialization of the snake's body length.

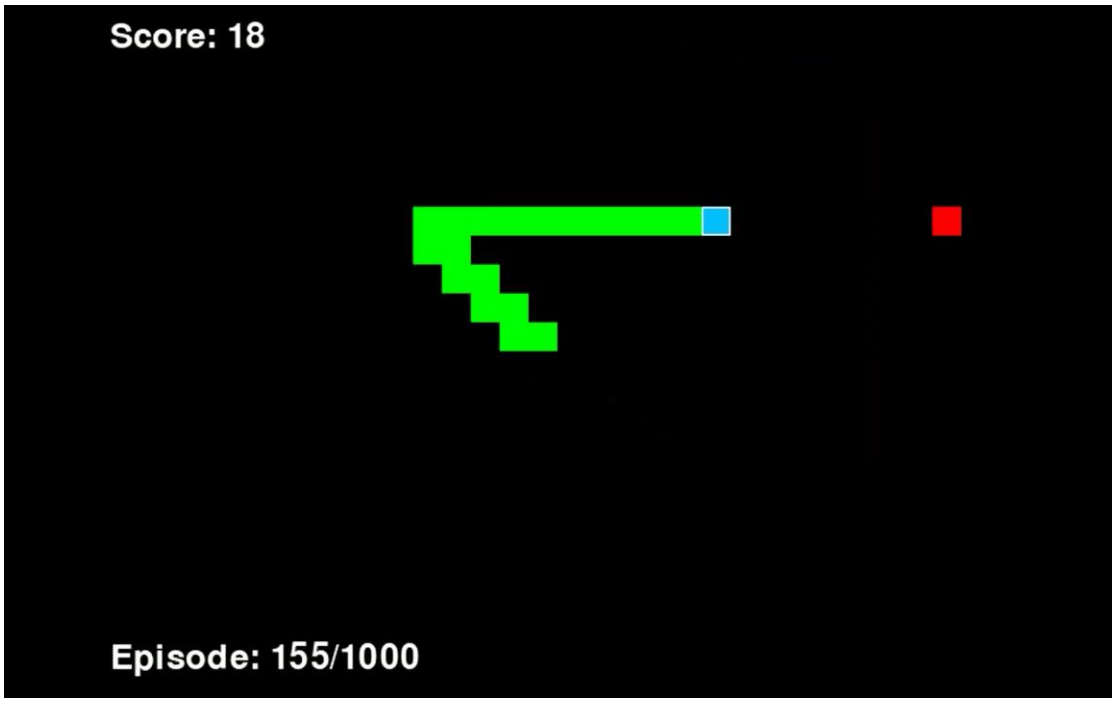

Figure 3.1 UI Introduction

3.3 Libraries and frameworks

Here are all the libraries and frameworks used in this project:

import numpy as np # is used to process state vectors and Q-value tables in Q-learning, providing efficient numerical calculations and array manipulation support

import pygame # is the main game development framework, providing graphical interface rendering, event handling, and game loop control

import random # is used to generate random snake initial directions, food locations, and random exploration in the ε-greedy strategy

from collections import deque # Use a double-ended queue data structure to efficiently manage the body position of snakes, supporting fast head and tail operations

Import time # is used to precisely control the frame rate of game and video recording, and realize the dynamic adjustment of game speed

import os # handles file and directory operations, and is used to create a save directory for videos and Q sheets, as well as to manage file paths

import cv2 # Processes video recordings of game footage, performs color space conversions, and saves video files of training and testing processes

import datetime # is used to generate a timestamp, create a unique file name for the saved file, and record the training test time

import json # is used to serialize and deserialize Q table data, and implement the saving and loading functions of Q table

3.4 User Interaction

In this project, we have set up the function of recording videos and the display of Q sheets. The display of the Q sheet can be viewed on the show.py.

Since the main purpose of this project is to use Q-learning for reinforcement learning of Snake, we do not provide users with the ability to operate the arrow keys.

4. Documentation and Presentation

4.1 Written Report

For game design in this document, please refer to:1. Game design

For more information about Q-learning, please refer to: 2. Q-Learning Implementation

The assessment results can be referred to: 2.2 Exploration and Exploitation and 2.3 Learning Policy

4.2 Video Presentation

Youtube Video Link: Snake-Agent

4.3 Challenges and Solutions

In the early stage of game development, I carried out a lot of data organization and game understanding. After reading a lot of github, blogs, papers and other materials, the learning route of Snake can be roughly summarized into the following four stages:

Stage 1 (Beginner), Stage 2 (Intermediate) , Stage 3 (Advanced) , Stage 4 (Perfection)

If judged by this standard, our snake should meet Stage 3, which means that handling simple situations is feasible, while handling complex situations needs improvement. I think this may be due to the fact that our state vectors are not rich enough, which makes its perception of the environment not perfect, for which I have made several attempts to optimize, with the most attempted being “Alley Detection”.

Since none of the attempted code improved the Agent's performance, but rather caused it to drop significantly, the code was not documented. However, I have drawn up a design (Figure 4.1) for ‘Alley’ detection that I hope will be useful for future work:

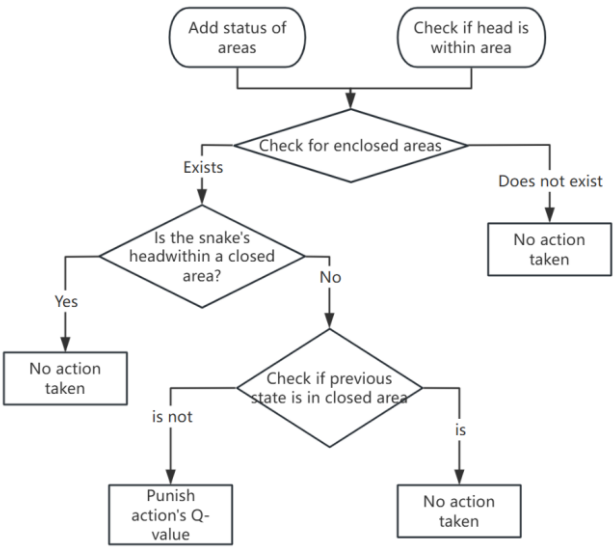

Figure 4.1 Enclosed Areas Logic Design