Reinforcement Learning Snake Game Design

CHAI Yaping

1 Game Design

Objective and Set of Rules

Objective The game's objective is for the snake to eat as much food as possible while avoiding collisions with the walls or its own body.

Rules The snake moves around the screen, eating food to grow longer. The game ends if the snake runs into the screen edges or itself.

State Space and Action Space

State Space This project defines the state by the relative position of the food to the snake's head, danger from the screen edges, and danger from the snake's own body. The state space is defined in the `paramsToState` function, which converts the game parameters, such as snake position, and food position into a state representation. It includes Relative food position, Screen danger or whether the snake is near the edge of the screen, Body danger or whether the snake is near its own body, and Current direction of the snake.

Action Space The action space consists of four possible actions Up (U), Left (L), Down (D), and Right (R). These actions are defined in the emulate function, where the agent chooses an action based on the Q-values or randomly for exploration.

Reward Function

Positive Rewards When the snake eats food, it receives a large positive reward `rewardScore = 50000000`.

Negative Rewards When the snake dies or collides with the wall or its own body, it receives a large negative reward `rewardKill = -10000`.

Small Penalties The snake receives a small negative reward `rewardAlive = -1` for each step it takes without eating food. This encourages the snake to find food quickly rather than wandering aimlessly.

Working Diagrams

Figure 1 contains three flowcharts that outline the logic of the Snake Game, implemented using reinforcement learning.

Game Flow The game starts by setting up the initial state, including the position of the snake, food, and game boundaries. The main loop of the game continues to execute until the game ends. It manages the gameplay by handling user inputs, updating the game state, and rendering the updated environment. Handle Input The system receives and processes input from the player. This could be direction changes or pausing the game. The snake's position is updated based on the current direction of movement, which is determined by the player's input. The system checks whether the snake has collided with itself or the walls. If a collision occurs, the game is over. The updated state of the game, namely the position of the snake, food, and any relevant game status is rendered on the screen for the player to view. If a collision has occurred, the game is over. The system will either reset the game or end it. If the game is not over, the loop repeats, allowing the game to continue.

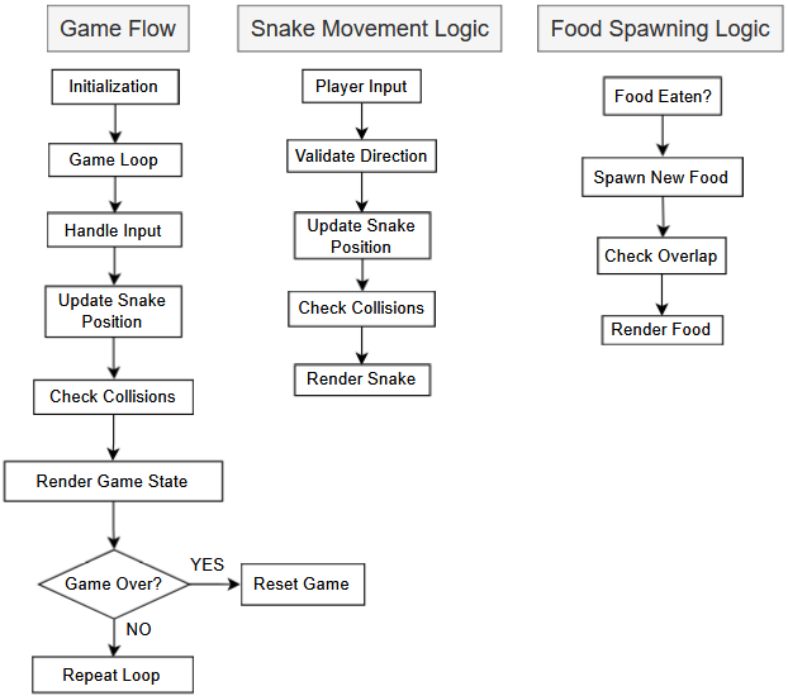

Figure 1: Three flowcharts of the Snake Game.

Snake Movement Logic The player provides input to control the direction in which the snake should move. The system checks whether the direction provided by the player is valid, i.e., the snake cannot reverse into itself. Based on the validated direction, the snake's position is updated. The updated position of the snake is checked for collisions with the wall or its own body. If a collision is detected, the game ends. The new position of the snake is rendered to reflect its movement on the screen.

Food Spawning Logic The system checks whether the snake has eaten food by checking if the snake's head overlaps with the food. If food is eaten, new food is spawned at a random location on the game board. The system ensures that the newly spawned food does not overlap with the snake or walls. The food is rendered at the new location on the screen.

2 Q-Learning Implementation

Q-Learning Algorithm Implementation

This project defines a function called `emulate`, which is the "brain" of the snake game. It makes decisions based on the current state of the game. It uses a Q-table to determine the best action to take. The Q-table is a dictionary, where the keys are states and the values are lists of 4 numbers, each representing the reward for taking a certain action (up, left, down, right) in that state. The function takes in a dictionary of parameters, including the snake's current state, the food position, and other relevant information. It returns the action to take: one of 'U', 'L', 'D', and 'R'.

The code below is a core part of the Q-learning algorithm. It updates the Q-value for a specific state-action pair based on the Bellman equation.

```
 # Update the previous reward for the previous state
prevReward[index] = (1 - alpha)  prevReward[index] + \
                    alpha  (reward + gamma  max(estReward) )
```

The updated Q-value is a weighted sum of:

1. The current Q-value ((1 - alpha) prevReward[index]), which represents the agent's prior knowledge.
2. The new information (alpha (reward + gamma max(estReward))), which represents the immediate reward and the discounted future reward.

The formula ensures that the Q-value is updated gradually, balancing the agent's prior knowledge with new information.

The code updates the Q-value for the previous state-action pair using the Q-learning update rule. It balances the agent's prior knowledge ((1 - alpha) prevReward[index]) with new information (alpha (reward + gamma max(estReward))). The update incorporates both the immediate reward and the discounted future reward, enabling the agent to learn the optimal policy over time. This update is a key part of the Q-learning algorithm, allowing the agent to improve its decision-making based on experience.

Policy Learning

Q-value Update on Game Over This project defines a callback function called `onGameOver(score, moves)`, which is called whenever the snake runs into itself or the border. It's responsible for updating the Q-table of the previous state which is the state that led to the game over.

When the game ends, the code below allows the agent to update the Q-value for the action that led to the failure. `rewardKill = -10000`, is a large negative value, penalizing the action severely and discouraging the agent from taking actions that lead to failure. By penalizing actions that lead to failure, the agent learns to avoid such actions, thereby increasing its survival time and cumulative rewards.

```
prevReward[index] = (1 - alpha)  prevReward[index] + \
                    alpha  rewardKill
```

Q-value Update on Eating Food

This project defines a function called `onScore(params)`, which gets called when the snake scores a point (eats the food). It updates the Q table with the new reward for the previous state and action.

When the agent eats food, the code below updates the Q-value for the action that led to eating food. `rewardScore = 50000000`, is a large reward for eating food, encouraging the agent to seek food. By rewarding actions that lead to eating food, the agent learns to seek food more effectively, thereby increasing cumulative rewards.

```
prevReward[index] = (1 - alpha)  prevReward[index] + \
                    alpha  (rewardScore + gamma  max(estReward))
```

Techniques Used

Epsilon-Greedy Exploration This project uses the parameter e to represent the probability of choosing a random action (exploration) versus the best-known action (exploitation). The parameter e_d is the decay factor for e. If the mode is "play", e is set to 0.0001 and e_d is set to 1. If the mode is "train", e is set to 0.9 and e_d is set to 1.3.

Learning Rate This project uses the parameter `alpha=0.1` as the learning rate to control how much new information overrides old information. The parameter `alphaD` is the decay factor for alpha.

Discount Factor This project uses the parameter `gamma=0.9` as a discount factor to determine the importance of future rewards.

3 Game Interaction

Creating the User Interface

This project uses `pygame.display.set_mode()` to create the game window, which sets up the visual interface for the game. This is where the environment (the game) is rendered and displayed to the user or agent.

The UI listens for user input (keyboard events) to control the snake's direction. This is done in the `mainGame` function using `pygame.event.get()`. The UI continuously updates and renders the game environment, including the snake, food, and score. This is done using `pygame.draw.rect()` and `game_window.fill()`. When the game ends, e.g., the snake collides with itself or the wall, the UI provides feedback by displaying a black screen and triggering the `onGameOver` callback. The UI controls the speed of the game using a frame rate controller `fps_controller.tick(FPS)`. This ensures that the game runs smoothly and provides a consistent experience for the user or agent.

UI Displaying

UI Displays the Current State The current state of the game, e.g., snake position, food position, and the score is visually rendered on the screen using `pygame.draw.rect()` and `game_window.fill()`. This provides a real-time representation of the environment.

UI Reflects Agent's Actions The agent's actions either from user input or AI decisions are reflected in the snake's movement. The direction of the snake is updated based on the agent's input, and the UI visually updates the snake's position.

UI Displays Rewards and Penalties Rewards and penalties are displayed through changes in the game state and score. The UI reflects rewards by updating the score and growing the snake, while penalties are shown by triggering the game over screen.

4 Evaluation

Evaluation Results

Functionality The core mechanics of the game, such as snake movement, food consumption, and growth, work as expected. The snake responds correctly to user input (arrow keys or WASD) and AI decisions. The game correctly detects collisions with walls and the snake's own body, triggering the game over state. The score increases appropriately when the snake eats food, and the game resets correctly after the game over, as illustrated in Figure 2.

User Interface The game window displays the snake and food clearly. The use of colors (green for the snake, red for the food, and black for the background) makes the game visually appealing and easy to understand. When the game ends, the screen clears, and the game resets, allowing for immediate replay.

User Experience The game is intuitive and easy to play, with clear controls and visual feedback. The game can be restarted instantly after a game over, encouraging multiple gameplay.

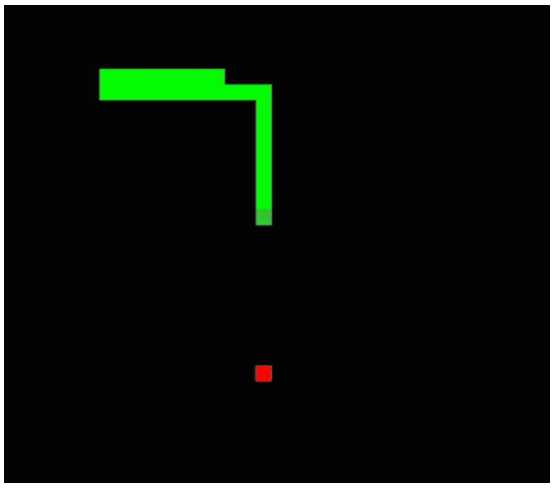

Figure 2: Three flowcharts of the Snake Game.

Challenges and Solutions

Handling User Input and Snake Movement It was difficult to ensure smooth and responsive control of the snake using keyboard inputs (arrow keys or WASD). The snake needed to change direction instantly but not reverse direction, e.g., from UP to DOWN in a single move. This project solved this challenge by implementing a direction and `change_to system` to queue direction changes. Besides, This project added validation to prevent the snake from reversing direction instantly.

Collision Detection It was difficult to detect collisions with the snake's own body and the walls required precise logic to avoid false positives or missed collisions. This project solved this challenge by implementing boundary checks for wall collisions and adding a loop to check if the snake's head collides with any part of its body.

Game Over and Reset It was difficult to reset the game state, e.g., snake position, score, and food position after a game over without restarting the entire program. This project solved this challenge by creating a `newGame()` function to reset the game state.