\

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Machine Leaning

Mini Project Report on

Autonomous Snake: Machine Learning-Based Game Control

SUBMITTED
BY

Amulya Parashar Skanda Sankar Raman Arhaan Girdhar

Section B

Under the Guidance of:

Dr. Narendra V G

Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal, Karnataka – 576104

October 2024

Abstract:

This project offers a novel take on the traditional Snake game, automating gameplay through the use of machine learning algorithms. The system uses a set of binary features taken from the game state to optimize the snake's movements using a Q-Learning algorithm. The project consists of a number of scripts that operate the snake, regulate the game's environment, and aid in the intelligent agent's training. The final objective is to show off the efficiency of machine learning in game automation while producing an enjoyable and captivating experience.

Keywords:

- Machine Learning
- Snake Game
- Q-Learning
- State Representation
- PyGame
- NumPy

Introduction:

A popular option for illustrating different programming ideas is the classic arcade game Snake. The goal of this project is to apply machine learning techniques to automate the Snake game. In order to maximise its score and prevent collisions, the goal is to develop an intelligent agent that can successfully traverse the game environment by making decisions depending on its present condition.

A collection of binary features that capture crucial elements of the environment, like the snake's length, direction, and food location, are used in this implementation to describe the game state. By traversing the game environment and updating its knowledge through trial and error, the intelligent agent uses a machine learning algorithm to learn the best course of action.

This project also demonstrates the integration of various Python modules, including NumPy for effective numerical computations and PyGame for the implementation of the game. By emphasising the agent's capacity for experience-based learning, this project underscores the value of state representation in decision-making processes and the promise of machine learning in creating intelligent gaming applications.

Literature Review:

In the fields of machine learning and artificial intelligence, there has been an increasing interest in applying machine learning approaches to classic games like the Snake game. Originally released on Nokia smartphones, the Snake game offers a straightforward yet difficult setting that may be used to test and illustrate a variety of machine learning techniques, such as Q-Learning. This study of the literature focusses on Q-Learning as a useful tool for automating game mechanics and investigates the creation and deployment of machine learning models in traditional gaming environments.

Classic Game Environments and Machine Learning

Classic games like Chess and Go were among the first settings in which machine learning agents were created, demonstrating the longstanding connection between machine learning and game theory. These kinds of games provide a controlled setting with clear regulations, which makes them perfect for evaluating the performance of different learning algorithms. The grid-based mechanics of Snake, along with the growing difficulty of avoiding the snake's body, make it an intriguing game despite its simplicity. In order to ascertain which machine learning techniques perform best in limited settings, researchers have employed game environments like Snake to investigate Q-Learning as well as alternative techniques like deep learning and evolutionary algorithms.

Machine Learning in Snake Game Automation

Numerous attempts have been made to automate the Snake game using machine learning. Neural networks, such Deep Q-Networks (DQN), which integrate Q-Learning and deep learning to maximise game play, are the subject of some research. In order to "evolve" the optimal approach for Snake, other research has experimented with evolutionary algorithms, including methods such as genetic algorithms. While these strategies have had some success, Q-Learning has been found to be a reasonably simple and efficient way to automate Snake games.

In 1989, Watkins created Q-Learning, an algorithm that updates a Q-table while the agent interacts with the environment in an attempt to maximize a cumulative reward. The Snake game's use of Q-Learning strikes a balance between ease of use and efficiency. Through trial and error, the algorithm learns, eventually improving the snake's motions by updating its understanding of the game world through feedback in the form of penalties (such as hitting with walls or its own body) and rewards (such as eating an apple). This concept gives the snake a way to modify its approach as the game goes on, resulting in a dynamic and more effective model for snake management.Q-Learning in Modern Applications

Despite not being a novel method, Q-Learning is still often employed because of its resilience and ease of use. Beyond game automation, Q-Learning is being used in modern industries like robots, banking, and driverless cars. Q-Learning is appropriate for situations such learning the Snake game, where the game grid can be divided into discrete cells and the snake's movements may be represented as discrete actions (up, down, left, and right). This is because Q-Learning can function in settings with discrete states and actions. With less computational needs than more sophisticated deep learning techniques, Q-Learning offers an approachable way to automate the Snake game by utilizing this algorithmic feature.

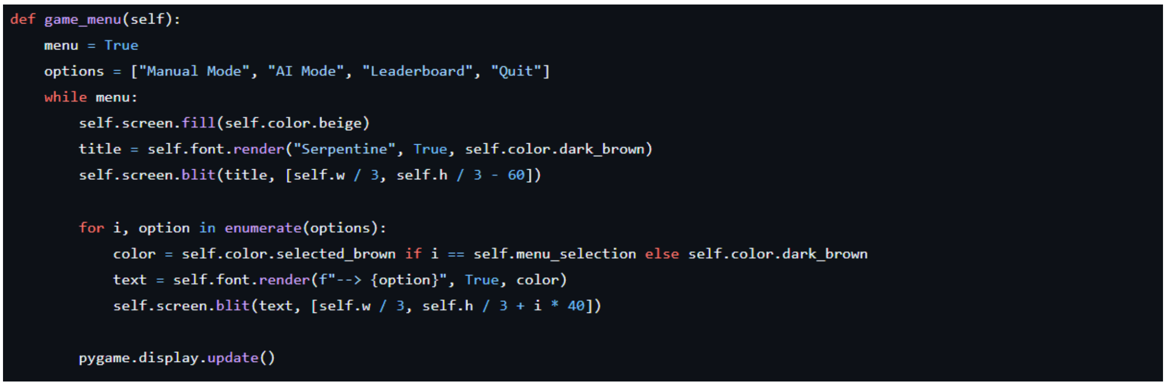
Conclusion

In conclusion, Q-Learning has shown itself to be a successful machine learning technique for automating gameplay in traditional settings such as Snake. According to the literature, Q-Learning is a popular option for researchers looking into game automation since it achieves the best possible mix between simplicity and performance, even if there are more sophisticated algorithms available. By effectively automating the Snake game, its use in this study advances the investigation of machine learning's possibilities in gaming environments and other contexts. A strong basis for expanding the algorithm's use in more intricate, grid-based contexts is provided by the corpus of current research on Q-Learning.

Methodology:

The project involves several steps to automate the Snake game using a Q-Learning agent. The methodology is structured as follows:

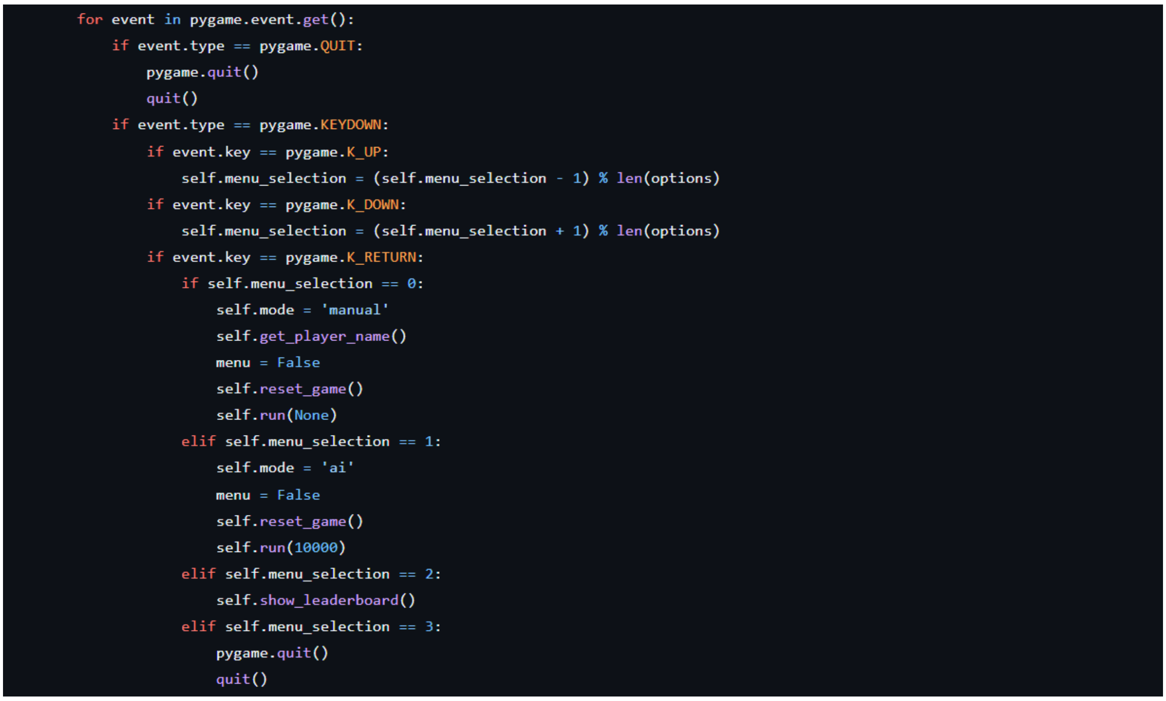
1. **Game Implementation:**
   The first step is to implement the Snake game using the PyGame library. PyGame provides the framework for creating the game environment, where the snake moves across the screen. The objective of the game is for the snake to collect food, which causes it to grow in length. The player must avoid collisions with the boundaries or the snake's own body, as either would result in the game ending. The game is set up with a grid-like environment, and the snake moves in defined directions (up, down, left, right). This setup forms the foundation of the environment in which the Q-Learning algorithm will later operate, learning optimal strategies for movement and maximizing the score by collecting food without hitting obstacles.



```python
def game_menu(self):
    menu = True
    options = ["Manual Mode", "AI Mode", "Leaderboard", "Quit"]
    while menu:
        self.screen.fill(self.color.beige)
        title = self.font.render("Serpentine", True, self.color.dark_brown)
        self.screen.blit(title, [self.w / 3, self.h / 3 - 60])

        for i, option in enumerate(options):
            color = self.color.selected_brown if i == self.menu_selection else self.color.dark_brown
            text = self.font.render(f"--> {option}", True, color)
            self.screen.blit(text, [self.w / 3, self.h / 3 + i * 40])

        pygame.display.update()
```



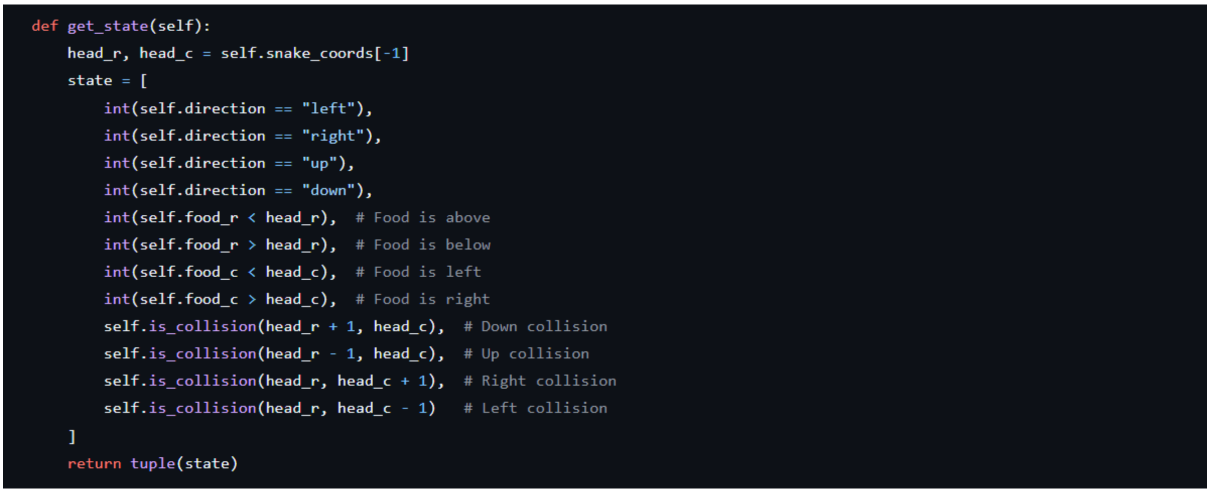
```python
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    self.menu_selection = (self.menu_selection - 1) % len(options)
                if event.key == pygame.K_DOWN:
                    self.menu_selection = (self.menu_selection + 1) % len(options)
                if event.key == pygame.K_RETURN:
                    if self.menu_selection == 0:
                        self.mode = 'manual'
                        self.get_player_name()
                        menu = False
                        self.reset_game()
                        self.run(None)
                    elif self.menu_selection == 1:
                        self.mode = 'ai'
                        menu = False
                        self.reset_game()
                        self.run(10000)
                    elif self.menu_selection == 2:
                        self.show_leaderboard()
                    elif self.menu_selection == 3:
                        pygame.quit()
                        quit()
```

2. State Representation:

The next step involves defining the state space for the Q-Learning agent. In this project, the state is represented by 12 binary features that capture the relevant information needed for decision-making. These features include:

- **Snake Position**: Coordinates of the snake's head.
- **Food Position**: Coordinates of the food.
- **Direction of Movement**: Current direction of the snake.
- **Proximity to Wall**: Indicators for walls on each side.

These features provide the necessary data for the agent to navigate the environment effectively.

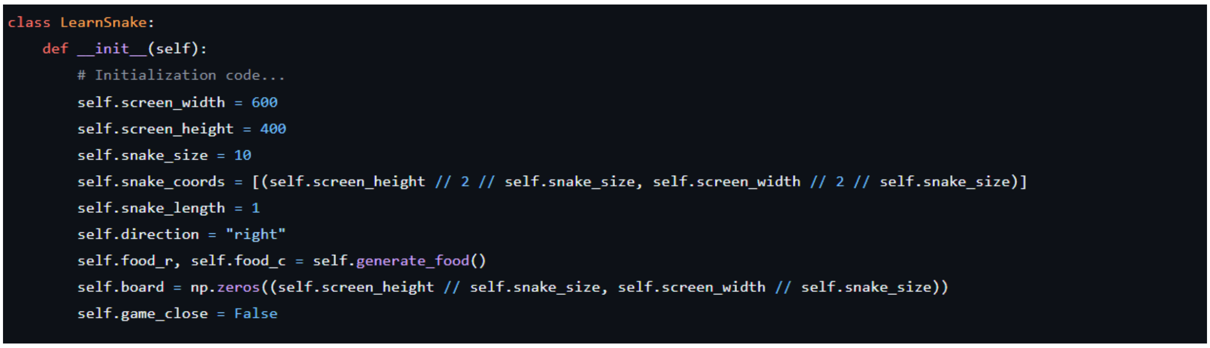


```
def get_state(self):
    head_r, head_c = self.snake_coords[-1]
    state = [
        int(self.direction == "left"),
        int(self.direction == "right"),
        int(self.direction == "up"),
        int(self.direction == "down"),
        int(self.food_r < head_r),  # Food is above
        int(self.food_r > head_r),  # Food is below
        int(self.food_c < head_c),  # Food is left
        int(self.food_c > head_c),  # Food is right
        self.is_collision(head_r + 1, head_c),  # Down collision
        self.is_collision(head_r - 1, head_c),  # Up collision
        self.is_collision(head_r, head_c + 1),  # Right collision
        self.is_collision(head_r, head_c - 1)   # Left collision
    ]
    return tuple(state)
```

3. Q-Learning Algorithm:

The Q-Learning algorithm enables the agent to learn an optimal policy by interacting with the game environment. A Q-table is used to store the learned values, mapping different state-action pairs. At each step, the algorithm selects an action based on the current state, either exploiting the highest known reward or exploring new possibilities. After executing the action, the environment responds with a reward, which the algorithm uses to update the Q-values in the table. Over time, the agent's policy improves as it maximizes the cumulative reward, achieving more effective gameplay.

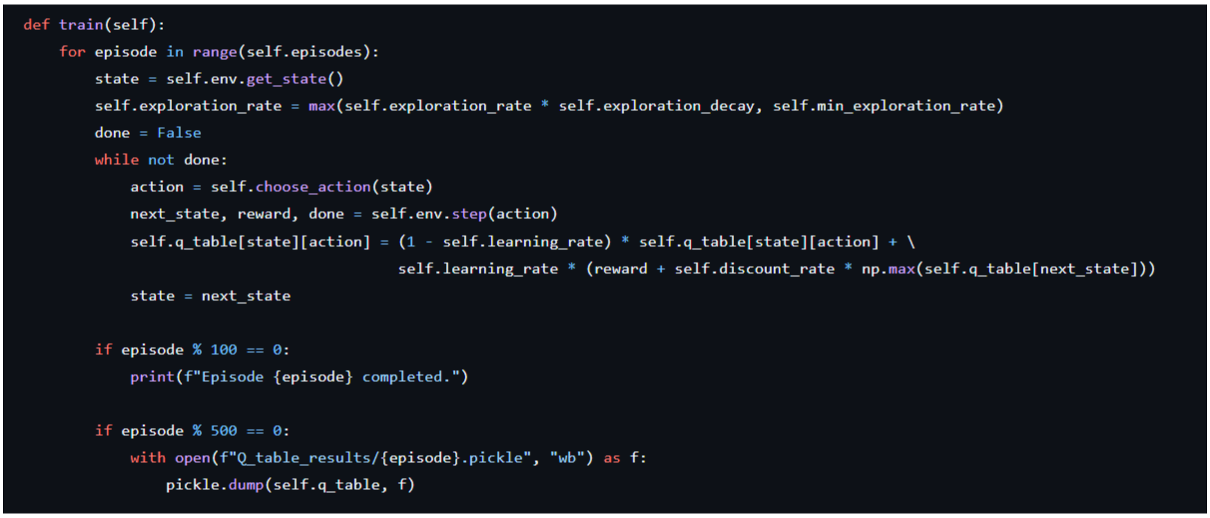
The snake’s movement and decision-making processes are driven by this learned policy, with the goal of continually increasing the score while avoiding collisions.



```
class LearnSnake:
    def __init__(self):
        # Initialization code...
        self.screen_width = 600
        self.screen_height = 400
        self.snake_size = 10
        self.snake_coords = [(self.screen_height // 2 // self.snake_size, self.screen_width // 2 // self.snake_size)]
        self.snake_length = 1
        self.direction = "right"
        self.food_r, self.food_c = self.generate_food()
        self.board = np.zeros((self.screen_height // self.snake_size, self.screen_width // self.snake_size))
        self.game_close = False
```

4. Training the Q-Learning Agent:

Training the agent involves running multiple episodes of the game, where the agent learns from its experiences. During each episode, the agent takes actions based on the current state, observes the reward, and updates the Q-table accordingly. The agent follows an epsilon-greedy strategy, balancing between exploration and exploitation. During exploration, the agent randomly selects actions to discover new strategies and outcomes. Meanwhile, exploitation occurs when the agent chooses actions based on the best-known values from the Q-table, aiming for optimal performance. Over time, this balance allows the agent to improve its decision-making abilities and learn the best moves for maximizing rewards.



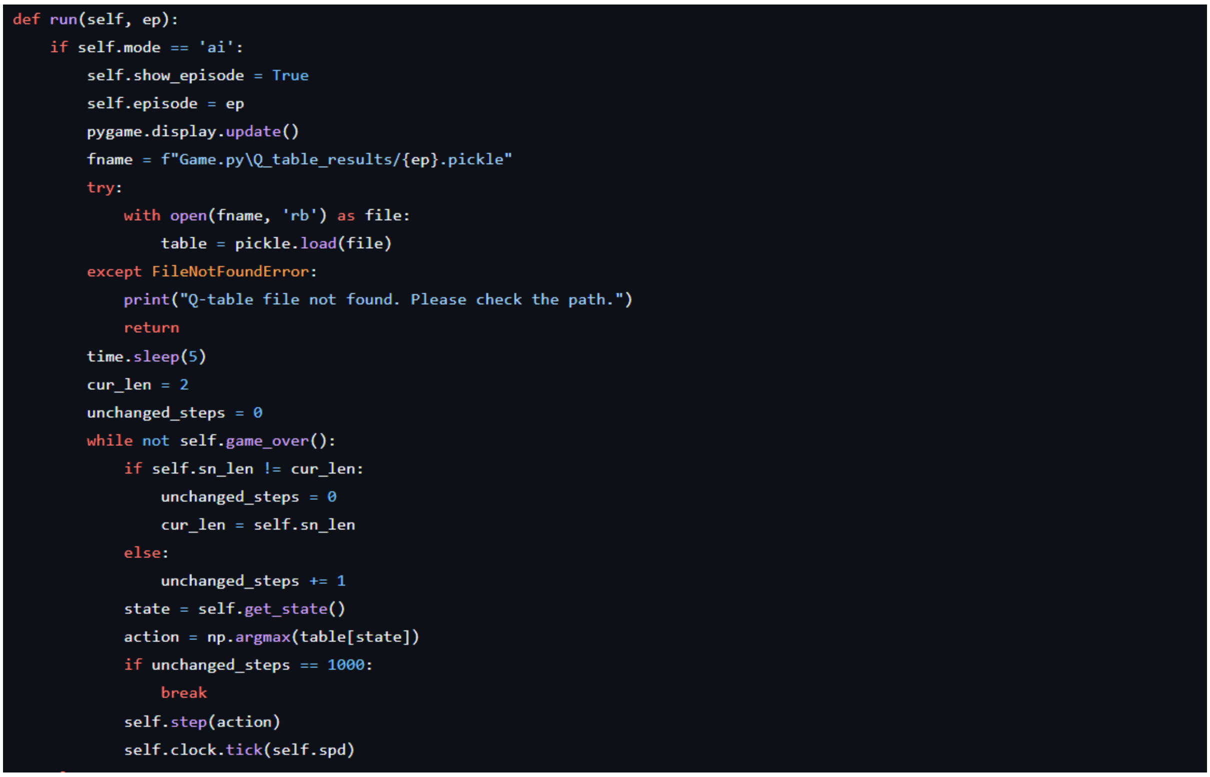
```
def train(self):
    for episode in range(self.episodes):
        state = self.env.get_state()
        self.exploration_rate = max(self.exploration_rate * self.exploration_decay, self.min_exploration_rate)
        done = False
        while not done:
            action = self.choose_action(state)
            next_state, reward, done = self.env.step(action)
            self.q_table[state][action] = (1 - self.learning_rate) * self.q_table[state][action] + \
                                          self.learning_rate * (reward + self.discount_rate * np.max(self.q_table[next_state]))
            state = next_state

        if episode % 100 == 0:
            print(f"Episode {episode} completed.")

        if episode % 500 == 0:
            with open(f"Q_table_results/{episode}.pickle", "wb") as f:
                pickle.dump(self.q_table, f)
```

5. Implementation of Trained Agent

After training, the performance of the Q-Learning agent is evaluated by running the game with the learned Q-table. The agent's ability to make optimal decisions based on its learned experiences is assessed through various metrics, such as the length of the snake and the number of food items collected.



```python
def run(self, ep):
    if self.mode == 'ai':
        self.show_episode = True
        self.episode = ep
        pygame.display.update()
        fname = f"Game.py\Q_table_results/{ep}.pickle"
        try:
            with open(fname, 'rb') as file:
                table = pickle.load(file)
        except FileNotFoundError:
            print("Q-table file not found. Please check the path.")
            return
        time.sleep(5)
        cur_len = 2
        unchanged_steps = 0
        while not self.game_over():
            if self.sn_len != cur_len:
                unchanged_steps = 0
                cur_len = self.sn_len
            else:
                unchanged_steps += 1
            state = self.get_state()
            action = np.argmax(table[state])
            if unchanged_steps == 1000:
                break
            self.step(action)
            self.clock.tick(self.spd)
```

Results and Discussions:

Results:

The implementation of the Q-Learning agent for the Snake game led to promising results after multiple training episodes. Initially, the agent struggled to perform efficiently, frequently colliding with walls or its own body. However, after several episodes of training, the Q-table was updated with values that helped the agent make better decisions. The following key observations were made:

- **Improvement Over Time:** As training progressed, the snake's performance improved significantly. It learned to avoid walls and its own body more effectively, and the number of apples collected per episode increased.
- **Epsilon-Greedy Strategy:** The balance between exploration and exploitation played a crucial role in the learning process. Early in training, exploration helped the agent discover strategies to avoid obstacles. In later stages, exploitation allowed the agent to consistently make the best decisions based on prior experiences.
- **Game Performance:** Towards the end of training, the agent demonstrated a higher success rate, with the snake collecting food consistently while avoiding collisions. The average score per game increased as a direct result of the optimized Q-values in the table.

Discussions:

The results indicate that the Q-Learning algorithm is a suitable approach for automating the Snake game, even though the simplicity of the problem limits the complexity of the state space and the learning process. By using 12 binary features to define the state space, the agent was able to learn relevant information such as the position of food, proximity to walls, and the direction of movement. The structure of the state space effectively captured the critical information needed to make decisions.

However, there are several points worth discussing regarding the agent's performance and potential limitations. One limitation of the Q-Learning algorithm in this implementation is its reliance on a discrete state space. The use of binary features simplifies the problem but may not capture more nuanced information about the game environment. For instance, the agent could have performed better with more detailed information about the distance to food or the snake's exact orientation relative to obstacles.

Another challenge lies in the exploration-exploitation trade-off. The epsilon-greedy strategy ensured that the agent explored new actions, but excessive exploration early on resulted in suboptimal decisions. Fine-tuning the epsilon decay rate could lead to faster convergence of the Q-values and improve the learning efficiency.

In terms of scalability, the agent’s performance is largely dependent on the size of the Q-table. As the game environment becomes more complex, the state space may grow exponentially, leading to increased memory requirements and slower learning times. More sophisticated techniques like function approximation using neural networks could address this issue, but they also introduce additional complexity to the algorithm.

Finally, the reward structure played a key role in the agent’s learning. Rewards were assigned based on food collection and collision avoidance, which helped the agent understand the primary objectives of the game. Adjusting the reward function could lead to different behaviours, potentially optimizing the agent’s strategy further.

In conclusion, the Q-Learning agent successfully learned how to play the Snake game and improved its performance over time. While the current implementation works well within the constraints of the problem, further enhancements could be made to refine the agent's decision-making abilities and adapt it to more complex environments.

Conclusion:

This project successfully demonstrates the application of the Q-Learning algorithm to automate the classic Snake game. Through a structured methodology that included game implementation, state representation, and the training of an intelligent agent, the project illustrated the potential of machine learning in enhancing gameplay. The agent learned to navigate the environment effectively, improving its performance over time by utilizing a well-defined set of binary features that captured crucial aspects of the game state.

The results highlighted significant advancements in the agent's decision-making capabilities, showcasing its ability to adapt and optimize strategies for maximizing the score while avoiding collisions. Despite the limitations related to the discrete state space and the challenges of the exploration-exploitation trade-off, the Q-Learning agent proved to be a robust approach for automating gameplay.

Future work could explore enhancing the agent's learning efficiency by fine-tuning the epsilon decay rate or integrating more sophisticated techniques, such as function approximation with neural networks. Additionally, experimenting with a more complex state representation could further optimize the agent's performance, making it adaptable to more intricate gaming environments. Overall, this project not only reinforces the viability of machine learning techniques in game automation but also contributes to the growing body of research focused on intelligent game agents.

References:

1. PyGame Tutorial - GeeksForGeeks: https://www.geeksforgeeks.org/pygame-tutorial/
2. Q-Learning Definition - TechTarget: https://www.techtarget.com/searchenterpriseai/definition/Q-learning
3. Q-Learning YouTube Tutorial: https://www.youtube.com/watch?v=TiAXhVAZQl8