**PHENIKAA UNIVERSITY  
 FACULTY OF ELECTRICAL AND ELECTRONIC ENGINEERING**

A logo with blue and orange circle

Description automatically generated

**Final Project Report for**

**Basic Reinforcement Learning**

**Topic: Apply Reinforcement Learning algorithms in snake game**

**Instructor's Name:** Vu Hoang Dieu   
**Student’s Name:** Nguyen Tien Thanh

**Student ID:** 22011103 **Class:** AI&RB K16

***Submission Date:*** *October 10, 2024*

**Table of Contents**

[**1.** **Introduction** 3](#_Toc179487923)

[**2.** **Theoretical Background** 4](#_Toc179487924)

[**2.1.** **Reinforcement Learning Basics** 4](#_Toc179487925)

[**2.2.** **Algorithms Overview** 4](#_Toc179487926)

[**2.2.1.** **Q-Learning** 4](#_Toc179487927)

[**2.2.2.** **SARSA** 5](#_Toc179487928)

[**2.2.3.** **Monte Carlo Methods** 5](#_Toc179487929)

[**2.2.4.** **Temporal-Difference Learning** 5](#_Toc179487930)

[**2.2.5.** **Summary** 5](#_Toc179487931)

[**3.** **Methodology** 7](#_Toc179487932)

[**3.1.** **Problem Definition** 7](#_Toc179487933)

[**3.1.1.** **State Space** 7](#_Toc179487934)

[**3.1.2.** **Action Space** 7](#_Toc179487935)

[**3.1.3.** **Reward Structure** 7](#_Toc179487936)

[**3.2.** **Algorithm Selection** 8](#_Toc179487937)

[**3.3.** **Implementation** 9](#_Toc179487938)

[**4.** **Experimental Setup** 11](#_Toc179487940)

[**4.1.** **Environment Setup** 11](#_Toc179487941)

[**4.2.** **Hyperparameters** 11](#_Toc179487942)

[**4.3.** **Hyperparameter Tuning** 11](#_Toc179487943)

1. **Introduction**

Within the discipline of machine learning, Reinforcement Learning (RL) examines how agents behave in a given environment to maximize cumulative rewards. This approach involves the agent learning optimal behavior through trial and error while interacting with its surroundings. RL is essential for developing intelligent systems capable of making decisions in dynamic and complex environments, such as gaming, autonomous vehicles, and robotics. Its significance lies in the ability to address problems where explicit programming is infeasible, enabling systems to adapt and enhance their performance over time.

This project focuses on developing an RL agent to play the classic Snake game, where the player controls a snake that moves around a board to eat food pellets. As the snake consumes food, it grows longer, increasing the challenge of navigating without colliding with itself.

The objective is to implement and compare various traditional RL algorithms, specifically Q-Learning, SARSA, Monte Carlo methods, and Temporal-Difference Learning, to effectively solve the Snake game. By analyzing the performance of these algorithms, the project aims to identify their strengths and weaknesses in navigating the game, consuming food, and avoiding collisions.

In the context of the Snake game, the report offers an in-depth review of various reinforcement learning techniques. It begins with a theoretical backdrop that includes detailed descriptions of each algorithm as well as basic notions such agents, environments, states, actions, and rewards. The experimental setup, including the Snake gaming environment, implementation details, and evaluation measures, is described in the methods section. Performance comparisons are used to present the findings, with an emphasis on the efficiency and rate of convergence of each approach.

1. **Theoretical Background**
   1. **Reinforcement Learning Basics**

Reinforcement Learning (RL) involves several core components that define how agents interact with their environments:

* **Agent**: The learner or decision-maker that takes actions to achieve a goal.
* **Environment**: The external system with which the agent interacts, providing feedback based on the agent's actions.
* **State**: A representation of the current situation of the agent in the environment.
* **Action**: The choices available to the agent that influence the environment.
* **Reward**: A scalar feedback signal that the agent receives after taking an action, indicating the immediate benefit of that action.
* **Policy**: A strategy that defines the agent's behavior, mapping states to actions.
  1. **Algorithms Overview**
     1. **Q-Learning**

Q-Learning is a model-free, off-policy algorithm that learns the value of actions in states. It aims to find the optimal policy by updating the Q-values (action-value function) using the Bellman equation:

**Q-Value Update:**

Where:

* α is the learning rate
* r is the reward received after taking action 𝑎 in state 𝑠
* s′ is the resulting state after the action
  + 1. **SARSA**

SARSA (State-Action-Reward-State-Action) differs from Q-Learning in that it is an on-policy algorithm, updating the Q-values based on the action actually taken by the agent. The update rule for SARSA is:

**Q-Value Update:**

where:

* a′ is the action chosen in state s′ according to the policy.
  + 1. **Monte Carlo Methods**

Monte Carlo methods learn value functions from complete episodes of experience. The key idea is to average the returns received for each state-action pair over many episodes to estimate their values:

* + 1. **Temporal-Difference Learning**

Temporal-Difference (TD) Learning combines ideas from Monte Carlo methods and dynamic programming. It updates estimates based on other learned estimates without waiting for the final outcome. The TD update rule is:

where V(s) is the value of state s.

* + 1. **Summary**

This section has outlined the fundamental concepts of reinforcement learning, including key components and the MDP framework. It has also provided an overview of critical algorithms—Q-Learning, SARSA, Monte Carlo methods, and Temporal-Difference Learning—along with their mathematical formulations for updating values. Understanding these foundations is essential for implementing and comparing these algorithms in the context of the Snake game.

1. **Methodology**
   1. **Problem Definition**

The problem tackled in this project is to develop a Reinforcement Learning agent capable of playing the classic Snake game. In this game, the agent controls a snake that moves around a grid to consume food pellets, growing longer with each pellet consumed while avoiding collisions with itself and the walls.

* + 1. **State Space**

The state space consists of all possible configurations of the game board. Each state can be represented as a grid where:

* The position of the snake is indicated by its coordinates.
* The position of the food is represented as another coordinate.
* The state may include additional information, such as the direction of the snake's movement.

For example, a state could be represented as a tuple containing the snake's head position and the food's position, along with the length of the snake.

* + 1. **Action Space**

The action space defines the possible movements the snake can make. In the Snake game, the actions typically include:

* Move Up
* Move Down
* Move Left
* Move Right

These actions enable the snake to change its position on the grid.

* + 1. **Reward Structure**

The reward structure in the Snake game is designed to further refine the agent's learning process by incorporating additional feedback based on proximity to food:

* **Positive Reward (+30)**: Awarded for consuming food, incentivizing the agent to seek out and eat food pellets.
* **Negative Reward (-50)**: Penalizes the agent for colliding with the walls or itself, indicating a loss and discouraging harmful actions.
* **Small Positive Reward (+0.01)**: Given when the agent moves closer to the food, encouraging efficient navigation toward the target.
* **Small Negative Reward (-0.5)**: Applied when the agent moves away from the food, discouraging inefficient movements that increase the distance to the target.
* **Neutral Reward (0)**: Assigned for all other actions that do not result in food consumption or collisions.

This comprehensive reward structure incentivizes the agent to consume food, avoid collisions, and optimize its path toward food, enhancing overall gameplay performance.

* 1. **Algorithm Selection**

For this project, specific algorithms were selected based on their suitability for the task:

* **Q-Learning**: Chosen for its off-policy nature, which allows the agent to learn from past experiences without following the current policy. This flexibility is beneficial for exploring different strategies in the Snake game.
* **SARSA**: Selected as an on-policy method to evaluate how the agent learns based on the actions it actually takes. This can provide insights into the performance and stability of the learning process.
* **Monte Carlo Methods**: Included to explore how learning from complete episodes affects performance, particularly in environments where immediate feedback is limited.
* **Temporal-Difference Learning**: Chosen for its ability to update value estimates based on other estimates, allowing for more efficient learning in environments like the Snake game where states are revisited frequently.
  1. **Implementation**

**Step-by-Step Procedure**

The Snake game environment was implemented from scratch using Python. The game logic includes grid management, snake movement, food spawning, and collision detection. Libraries used include:

* **pygame**: A library for creating games in Python, used for rendering the game graphics, handling user input, and managing game states.
* **random**: A standard Python library for generating random numbers, used for placing food randomly on the game board.
* **numpy**: A library for numerical computing in Python, utilized for efficient array operations and mathematical calculations.
* **collections.defaultdict**: A specialized dictionary that provides default values for non-existent keys, used for managing Q-values in a flexible manner.
* **pickle**: A Python module for serializing and deserializing Python objects, used for saving and loading the trained model.
* **matplotlib.pyplot**: A plotting library for Python, used for visualizing the performance metrics and learning curves of the reinforcement learning agent.
* **snake**: A custom module (presumably defined in your project) that contains the implementation of the Snake game logic, including game mechanics and state management.

For the algorithm implementation:

* **Q-Learning**: The Q-table is initialized with zeros. For each episode, an action is chosen using an epsilon-greedy strategy, the reward is observed, and Q-values are updated using the Q-learning update rule.
* **SARSA**: Similar to Q-Learning, but the action is chosen based on the current policy, and Q-values are updated using the SARSA update rule.
* **Monte Carlo Methods**: Complete episodes are run, storing state-action-reward sequences and updating the value estimates based on the returns.
* **Temporal-Difference Learning**: TD updates are utilized to refine value estimates as the agent interacts with the environment.

Training involves running each algorithm over multiple episodes, recording performance metrics such as average score, convergence rate, and stability. Visualizations are created to compare the learning curves of each algorithm.

This methodology systematically approaches the problem of training an RL agent to play the Snake game, leveraging established algorithms and structured implementation for effective learning and performance evaluation.

* + - * 1. **Experimental Setup**
  1. **Environment Setup**

The experimental configuration involves a custom environment designed for the Snake game, which allows the Reinforcement Learning agent to interact with the game mechanics effectively. The environment includes features such as grid management, snake movement, food spawning, and collision detection.

* 1. **Hyperparameters**

Several key hyperparameters were defined for the training of the Reinforcement Learning algorithms:

* **Learning Rate (α)**: Set to **0.01**, this parameter controls how much the agent updates its Q-values based on new experiences. A moderate learning rate balances speed of learning and stability.
* **Discount Factor (γ)**: Set to **0.99**, this factor determines the importance of future rewards. A value close to 1 encourages the agent to consider long-term rewards, which is crucial for strategic decision-making in the Snake game.
* **Exploration Rate (ε)**: Initialized at **1.0** (100% exploration), this rate dictates the likelihood of the agent selecting a random action versus the best-known action. This rate is gradually decreased (epsilon decay) to encourage more exploitation of learned strategies over time.
* **Number of Episodes**: The agent is trained over **10000 episodes**. Each episode consists of the agent playing the game until it collides or consumes food, allowing it to gather sufficient experience for learning.
  1. **Hyperparameter Tuning**

The process of tuning hyperparameters involved several steps:

1. **Initial Setup**: Starting with default values for the learning rate, discount factor, and exploration rate, the agent was trained using Q-Learning to establish a baseline performance.
2. **Exploration Rate Decay**: A decay strategy for the exploration rate was implemented. The ε value was gradually reduced from 1.0 to 0.1 over the first 1000 episodes, allowing the agent to explore initially and slowly shift to exploiting its learned knowledge.
3. **Learning Rate Adjustment**: The learning rate was experimented with values between **0.01** and **0.5**. After testing various configurations, a value of **0.1** was found to provide a good balance between learning speed and stability.
4. **Discount Factor Testing**: The discount factor was tested with values of **0.9**, **0.95**, and **0.99**. A higher discount factor (0.99) was chosen as it led to better performance in terms of long-term strategy.
5. **Performance Evaluation**: After tuning the hyperparameters, the performance of the agent was evaluated based on metrics such as average score per episode, convergence rate, and stability of learning curves.

Through this iterative process, the hyperparameters were refined to optimize the learning effectiveness of each algorithm, ensuring that the RL agent could navigate and perform well in the Snake game environment.

1. **Results**
   1. **Results Presentation**

Ảnh có chứa văn bản, ảnh chụp màn hình, Sơ đồ, hàng

Mô tả được tạo tự độngẢnh có chứa văn bản, ảnh chụp màn hình, Sơ đồ, hàng

Mô tả được tạo tự độngThe results of the experiments conducted for each Reinforcement Learning algorithm (Q-Learning, SARSA, Monte Carlo Methods, and Temporal-Difference Learning) are presented below.

Figure 2Monte-carlos rewards and food eaten over episodes

Figure 1: Q-learning rewards and food eaten over episodes

Ảnh có chứa văn bản, ảnh chụp màn hình, Sơ đồ, biểu đồ

Mô tả được tạo tự độngẢnh có chứa văn bản, ảnh chụp màn hình, Sơ đồ, hàng

Mô tả được tạo tự động

Figure 3 SARSA rewards and food eaten over episodes

Figure 1 Temporal-Difference Learning rewards and food eaten over episodes

## **Results Summary**

* **SARSA**:
  + Mean rewards after 10,000 episodes: **115.30**
  + Mean food eaten after 10,000 episodes: **5.57**
* **Temporal-Difference Learning (TD)**:
  + Mean rewards after 10,000 episodes: **116.33**
  + Mean food eaten after 10,000 episodes: **5.60**
* **Monte Carlo**:
  + Mean rewards after 10,000 episodes: **5.60**
  + Mean food eaten after 10,000 episodes: **2.11**
* **Q-Learning**:
  + Mean rewards after 10,000 episodes: **118.20**
  + Mean food eaten after 10,000 episodes: **5.67**

**Analysis**

1. **Q-Learning** achieved the highest mean rewards and the most food consumed, indicating effective learning and optimal policy discovery. Its off-policy nature allows it to learn from all experiences, contributing to better exploration and reward maximization.
2. **Temporal-Difference Learning (TD)** showed competitive results, slightly behind Q-Learning. Its ability to update estimates based on other estimates helps maintain stability but may limit exploration compared to Q-Learning.
3. **SARSA** produced similar results to TD but was marginally less effective. As an on-policy method, it is directly influenced by the agent's current policy, which can lead to slower learning in complex environments.
4. **Monte Carlo** performed significantly worse than the other methods, yielding low rewards and food consumption. This is likely due to its reliance on complete episode learning, which can be less efficient in environments with many possible states and actions.

**Conclusion**

Overall, the experiments reflect the importance of algorithm selection based on the problem's nature and complexity. Q-Learning stands out as the most effective method for this particular task, while the other algorithms provide valuable insights into reinforcement learning dynamics. Future work could involve exploring more complex environments or integrating advanced techniques like deep reinforcement learning to enhance learning capabilities.