Reinforcement Learning in Maze Environment

Abstract

The field of Reinforcement Learning (RL) offers diverse methodologies to solve decision-making problems, where agents learn optimal behaviors through interactions with an environment. This project explores the application of two distinct RL techniques: Deep Q-Network (DQN) and Basic Q-Learning (BQN), in a simulated maze environment. Our environment, designed in Python, challenges an agent to navigate from a start point to a destination while avoiding traps. The agent's journey through various difficulty levels (easy, medium, and hard) offers insights into the effectiveness and adaptability of both algorithms.

The DQN model, integrating neural networks, and the BQN model, employing a simple Q-table, were developed to understand their learning patterns and efficiency in the given environment. This report documents the implementation details, challenges faced, and the experimental setup, which included varied iterations and memory sizes. The results, primarily focusing on the rate of learning and the total reward per episode, provide a comparative analysis of both models. This study aims to contribute to the understanding of RL application in problem-solving and offers a foundation for future enhancements in RL techniques within simulated environments.

Introduction

Overview of Reinforcement Learning:

Reinforcement Learning (RL) is a pivotal area of machine learning where an agent learns to make decisions by interacting with an environment. The objective is to develop a strategy that maximizes

cumulative rewards over time. RL differs from other machine learning paradigms by its focus on learning from the consequences of actions, rather than from direct data labeling.

Significance in Simulated Environments:

Simulated environments, such as the maze used in this project, offer controlled settings to study the behavior of RL agents. These environments allow us to examine how agents learn and adapt to achieve goals, providing valuable insights into the mechanisms of learning and decision-making.

Project Objectives:

This project aims to implement and compare two distinct RL techniques: Deep Q-Network (DQN) and Basic Q-Learning (BQN). The chosen maze environment poses challenges like navigation and trap avoidance, which test the agents' ability to learn and adapt. The primary objectives of this project are:

1. To develop a versatile maze environment in Python that can adjust in complexity.
2. To implement DQN and BQN models and integrate them into this environment.
3. To analyze and compare the learning efficiency and adaptability of these two models in varying levels of maze complexity.
4. To identify the strengths and limitations of each model in the context of problem-solving in RL.

Theoretical Background

Fundamentals of Reinforcement Learning:

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions by trial and error, receiving feedback in the form of rewards. In RL, an agent interacts with its environment in discrete time steps. At each time step, the agent receives the environment's state, selects an action,

and receives a reward along with the new state. The goal is to learn a policy that maximizes the expected cumulative reward.

Q-Learning

Q-Learning is a model-free RL algorithm that seeks to find the best action to take given the current state. It's a value-based method that maintains a table (Q-table) where it stores the expected utility of taking a given action in a given state. The Q-values are updated using the Bellman equation, allowing the agent to improve its policy iteratively.

Deep Q-Network (DQN)

Deep Q-Network (DQN) is an extension of Q-Learning, where deep learning is integrated to approximate the Q-value function. DQN uses a neural network to predict Q-values, which is particularly useful in environments with large state spaces where maintaining a Q-table is impractical. The key innovations in DQN, such as experience replay and fixed Q-targets, have significantly improved the stability and performance of neural network-based Q-learning.

While both DQN and BQN follow the same fundamental principle of maximizing cumulative rewards, their approaches differ significantly. DQN leverages the power of neural networks for function approximation, making it suitable for complex environments with high-dimensional state spaces. In contrast, BQN is more straightforward, relying on a Q-table, and is typically used in simpler, discrete spaces. In the context of a maze environment, these algorithms offer distinct approaches to learning and navigating. The maze's structure, with its traps and varying difficulty levels, provides an excellent testbed to evaluate the effectiveness of DQN and BQN in terms of learning efficiency, adaptability, and scalability.

Implementation

Overview

The implementation of the project revolves around creating a functional maze environment and integrating it with two Reinforcement Learning models: Deep Q-Network (DQN) and Basic Q-Learning (BQN). The project is developed in Python, leveraging libraries such as Keras for DQN and NumPy for general operations.

Environment Setup

The maze environment (`Env` class) is a grid of varying sizes based on the difficulty level. It features key components like the starting point, destination, and traps. The complexity of the environment is a crucial factor, as it tests the adaptability and efficiency of the RL models.

- Grid Size and Complexity: The grid size changes with the level, ranging from a 4x4 grid for 'easy' to a 10x10 grid for 'hard'. This variability challenges the agent with more complex paths and decisions as the difficulty increases.

- Point Generation: The start, trap, and destination points are randomly generated for each episode, ensuring that the agent encounters different scenarios in each run.

Reinforcement Learning Models

Two distinct models are implemented to navigate this environment:

- Deep Q-Network (DQN): This advanced model uses a neural network to estimate Q-values. The implementation involves designing a network architecture suitable for processing the state inputs (the agent's position in the grid) and outputting a value for each possible action. The DQN's ability to handle high-dimensional state spaces makes it ideal for more complex mazes.

- Basic Q-Learning (BQN): A more straightforward approach compared to DQN, BQN utilizes a Q-table to store and update the values. It's a model-free algorithm that learns the value of actions in each state through trial and error, making it simpler but less scalable for larger state spaces.

The project is designed to be user-friendly, allowing users to specify parameters through command-line arguments. These parameters include the maze's difficulty level, the number of iterations for training the models, and the choice between DQN and BQN.

User Interaction: Users can dynamically interact with the project by running commands such as `python file_name.py --level medium --iteration 100 --model dqn`. This flexibility allows for extensive experimentation with different settings and models.

A significant part of the implementation involved tuning parameters like the learning rate, discount factor, and exploration rate. Balancing exploration (trying new actions) and exploitation (using known information) is vital for the effectiveness of the models. Additionally, ensuring that the DQN model accurately interprets the state inputs from the grid-based maze was a complex task, requiring several iterations of design and testing.

Integration and Validation

The final step involved integrating the RL models with the maze environment and validating their performance. This process required rigorous testing to ensure that the models could effectively learn and navigate the maze, adapting their strategies over time to maximize rewards.

Experimentation and Results

Experiment Setup

The experiments were designed to test the learning efficiency and adaptability of both DQN and BQN models in navigating the maze. The key parameters varied in the experiments included:

- **Difficulty Levels:** The maze was set to 'easy', 'medium', and 'hard' levels, impacting the grid size and complexity.
- **Number of Iterations:** Each model was run for a predefined number of iterations, allowing it to learn and adapt its strategy over time.
- **Model-Specific Parameters:** For DQN, parameters like learning rate, memory size, and target model update frequency were adjusted. For BQN, learning rate, discount factor, and exploration rate were the main focus.

Performance Metrics

The models were evaluated based on the following metrics:

- **Total Reward Per Episode:** This measures the cumulative reward the agent accumulates in each episode, providing insight into how well the agent learned to navigate the maze.
- **Steps to Reach the Goal:** The number of steps taken to reach the goal in each episode indicates the efficiency of the learned strategy.

- **Learning Progression:** Observing the change in total reward and steps over iterations helped assess how quickly and effectively each model learned.

Results

Analysis Across Difficulty Levels(Training Screenshots for Each Level)

Easy Level:

DQN Output: Quick adaptation with the agent achieving the goal in fewer steps.

BQN Output: Also showed adaptation, but with more steps compared to DQN, indicating less efficiency.

Medium Level:

DQN Output: The agent encountered more challenges, reflected by the longer paths and more negative rewards before reaching the goal.

BQN Output: Struggled with the increased complexity, with many steps and negative rewards indicating a less efficient learning process.

Hard Level:

DQN Output: Despite the complexity, the DQN showed an ability to learn a strategy to reach the goal with a stabilized reward pattern after an initial drop.

BQN Output: Showed the agent's fluctuating decision-making process, with a final successful outcome but a clear indication of inefficiency in pathfinding.

From easy to hard levels, the DQN model demonstrated superior adaptation and learning efficiency, with its performance showing less impact by increased complexity compared to the BQN model. The

BQN model, while still capable of eventually finding the goal, required significantly more steps and exhibited a less stable learning process, particularly at higher complexity levels. The difference in performance between the models became more pronounced as the difficulty level increased, highlighting the advantages of DQN's neural network approach in complex environments.

Graphical Representations

The experiments generated graphical outputs depicting the learning curves of both models. These graphs illustrated the models' progression in total rewards and steps taken across episodes, providing a visual representation of their learning efficiency.

Learning Curves: Displayed the gradual improvement or fluctuations in the models' performance over time.

Reward Trends: Highlighted the models' ability to maximize rewards and learn efficient navigation strategies.

Graphs for Each Level

Easy Level:

DQN Game History: The graph indicated a learning curve with significant early exploration, followed by stabilization in reward, suggesting the agent quickly adapted to the environment.

DQN Learning History: The learning history showed a sharp initial decrease in loss, demonstrating efficient pattern recognition and policy improvement.

Medium Level:

DQN Game History: More volatility in the reward pattern compared to the easy level, reflecting the increased complexity and the agent's struggle to adapt.

DQN Learning History: Higher and more frequent spikes in loss, indicating the agent encountered more complex situations that were harder to generalize.

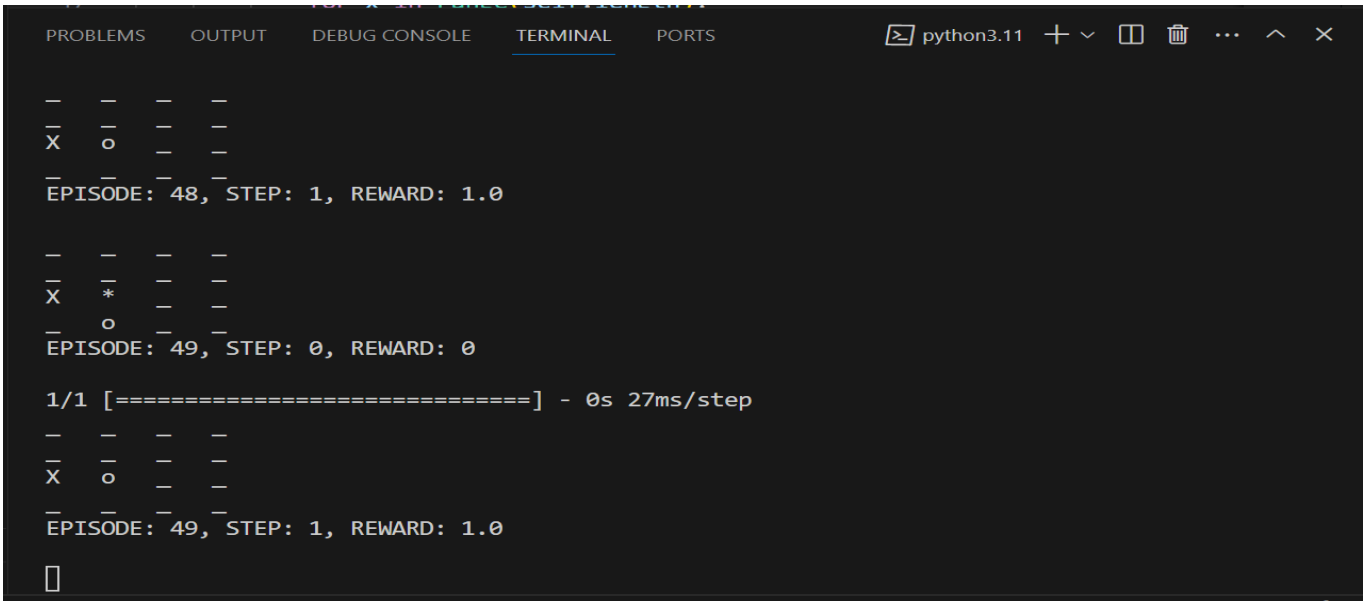
Hard Level:

DQN Game History: A steep initial drop in total reward followed by a quick stabilization, indicating a rapid adaptation to the complex environment.

DQN Learning History: Significant initial loss with volatility, suggesting a more challenging state space but with the model still able to learn an effective strategy over time.

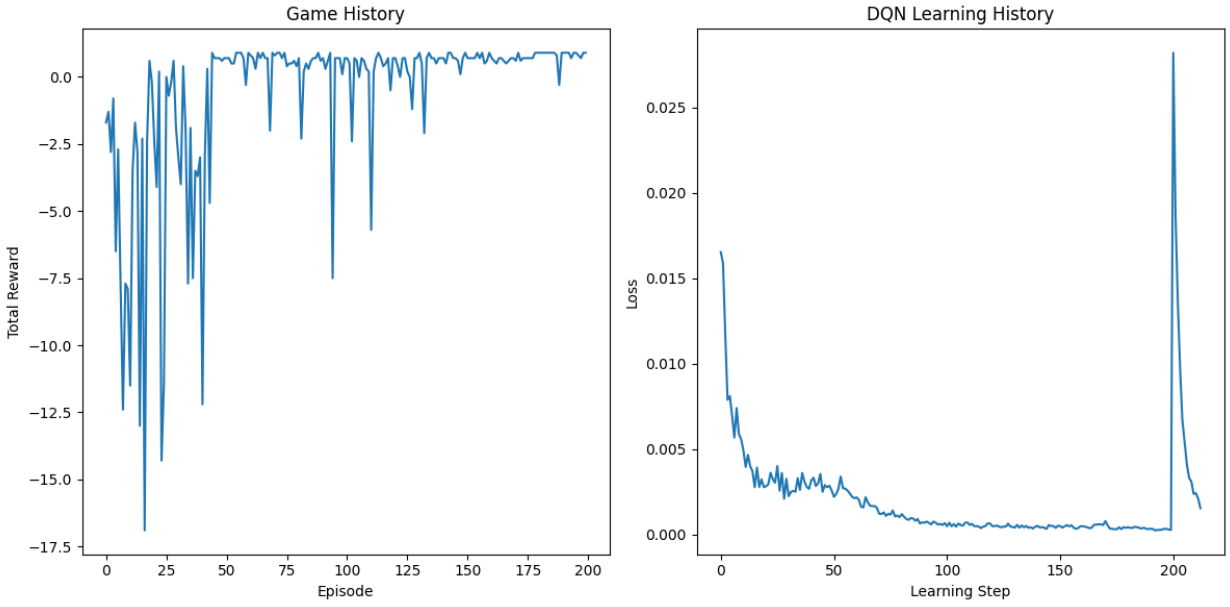
Github : https://github.com/venkateshterikuti/final-project-venkateshterikuti.git

For model: DQN, Level: easy, iteration: 50

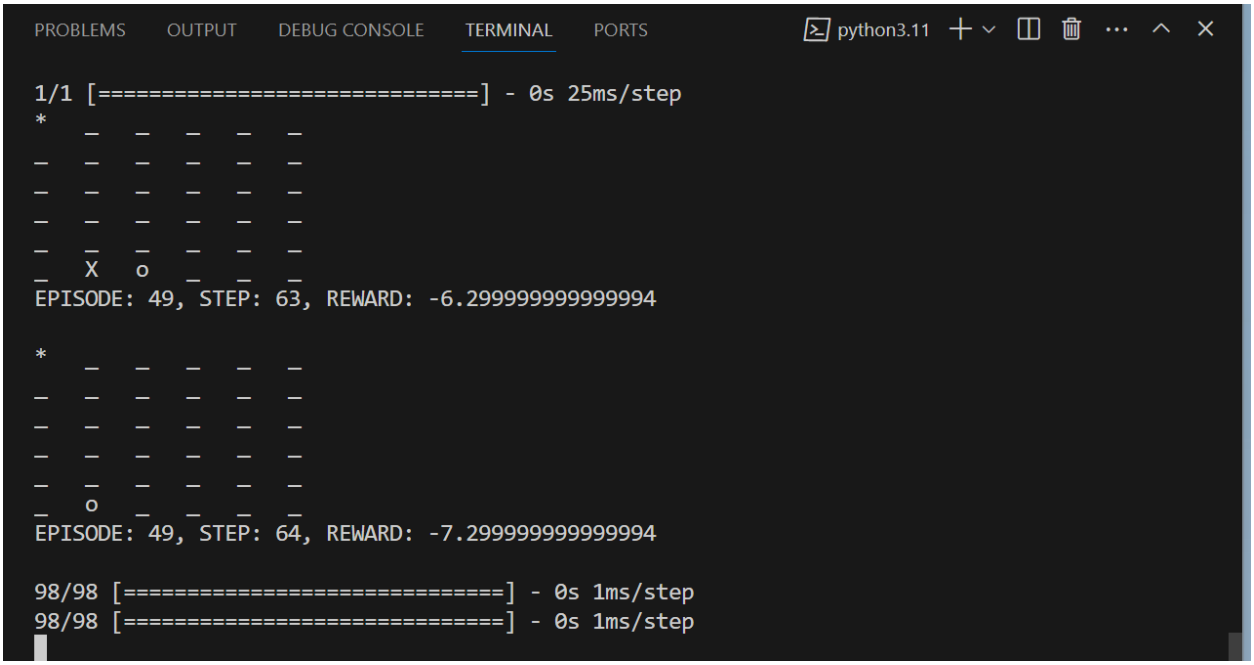


For model: BQN, Level: easy, iteration: 50

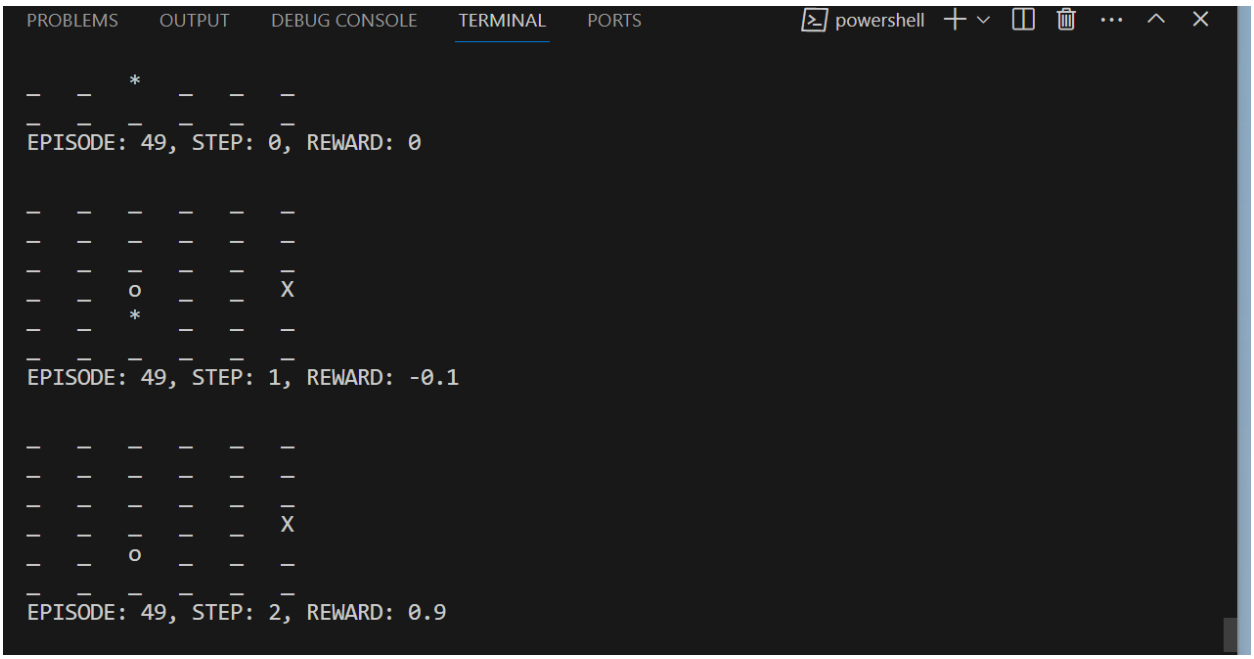
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS        powershell

EPISODE: 49, STEP: 1, REWARD: -0.1

_  _  X  *
_  _  o  _
_  _  _  _
_  _  _  _
EPISODE: 49, STEP: 2, REWARD: -0.2

_  _  X  *
_  _  _  o
_  _  _  _
_  _  _  _
EPISODE: 49, STEP: 3, REWARD: -0.30000000000000004

_  _  X  o
_  _  _  _
_  _  _  _
_  _  _  _
EPISODE: 49, STEP: 4, REWARD: 0.7
```

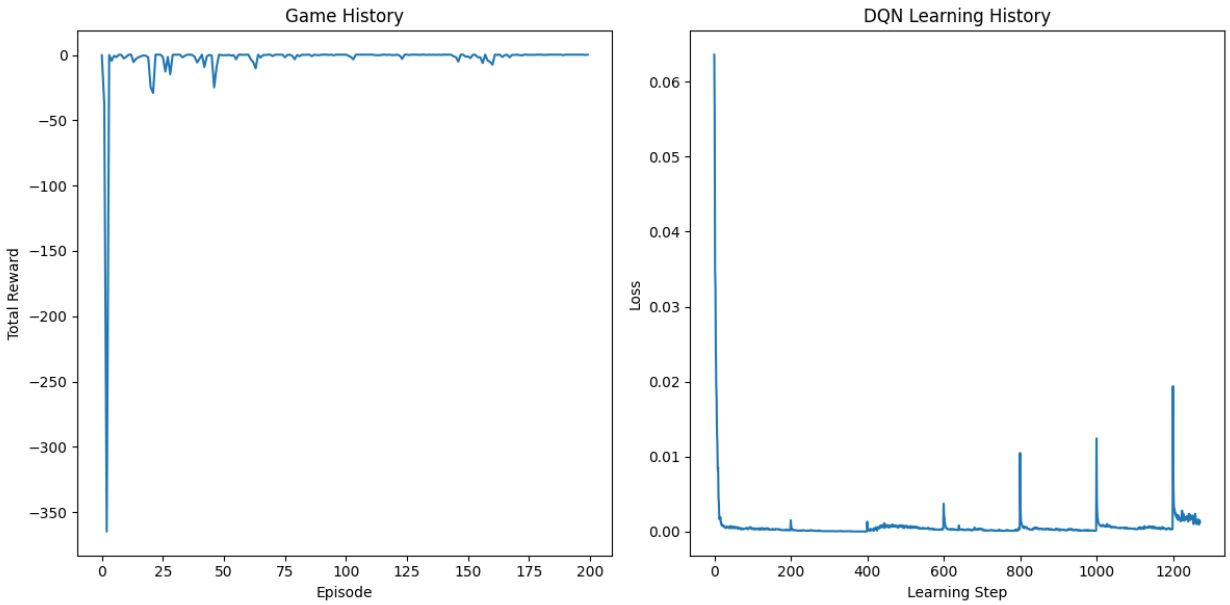


For model: DQN, Level: medium, iteration: 50



For model: BQN, Level: medium, iteration: 50





For model: DQN, Level: hard, iteration: 50

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS        powershell

_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  o  _
_  _  _  _  _  _  _  _  *  _
EPISODE: 49, STEP: 1, REWARD: -0.1

1/1 [==============================] - 0s 23ms/step
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  X  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  o  _
EPISODE: 49, STEP: 2, REWARD: 0.9

PS C:\Users\ual-laptop\VS code\maze\Final Project> \
                                  Select Language Mode
```

For model: BQN, Level: hard, iteration: 50

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                powershell

_  _  _  _  _  _  _  _  _  _
_  _  _  _  *  _  _  _  _  _
_  _  X  _  o  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
EPISODE: 49, STEP: 4, REWARD: -0.4

_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  o  _  _  _  _  _
_  _  X  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
_  _  _  _  _  _  _  _  _  _
EPISODE: 49, STEP: 5, REWARD: 0.6

Ln 1, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.11.7 64-bit (Microsoft Store)
```