**Aim**

The aim of this project is to implement a Q-learning algorithm to enable an AI agent to learn optimal strategies for playing the game of Ludo. This involves the AI agent improving its decision-making capabilities through experience and feedback in the form of rewards.

**Objective**

1. Develop an AI agent capable of playing Ludo using Q-learning.
2. Design and manage a state-action table that captures the value of different actions in different states.
3. Implement a reward system to guide the AI agent's learning process.
4. Train the AI agent over multiple game simulations to improve its performance.
5. Evaluate the performance of the AI agent in various game scenarios.

**Summary**

This project leverages Q-learning, a reinforcement learning algorithm, to create an AI agent that plays Ludo. By maintaining a Q-table to record the value of actions taken in different states, the agent learns which actions yield the highest rewards over time. The project includes defining the state and action space, initializing and updating the Q-table, and implementing a policy for action selection. The AI is trained through multiple episodes where it learns from the outcomes of its actions, gradually improving its gameplay strategy.

**Tools and Libraries Used**

- **VS Code**: Integrated Development Environment (IDE) for writing and debugging code.
- **Python**: Programming language used for implementing the Q-learning algorithm.
- **Numpy**: Library for numerical computations, used for creating and updating the Q-table.
- **Ludopy:**

  Description: A Python library tailored for implementing and simulating Ludo games. It provides tools for setting up the game board, managing game rules, and handling player interactions.

  Role: Facilitates the development and testing of Ludo game mechanics. It simplifies the process of creating the game environment and allows you to focus on integrating and optimizing the Q-learning algorithm for playing Ludo.

- **Matplotlib:**
  Description: A Python library tailored for implementing and simulating Ludo games. It provides tools for setting up the game board, managing game rules, and handling player interactions.

  Role: Facilitates the development and testing of Ludo game mechanics. It simplifies the process of creating the game environment and allows you to focus on integrating and optimizing the Q-learning algorithm for playing Ludo.

**Procedure (Explain the Code)**

1. **ActionTableEntry Class**:
    - **Initialization**: This class initializes with a piece and value, representing an entry in the action table.
    - **Add Entry Method**: Although not fully implemented, this method is designed to add entries to the action table.
2. **ActionTable Class**:
    - **Attributes**:
        - action_table: A numpy array storing state-action values.
        - piece_to_move: A numpy array storing the pieces to move for each state-action pair.
        - state: The current state of the game.
    - **Methods**:
        - __init__(self, states, actions): Initializes the action table with the specified number of states and actions, and calls the reset method to initialize arrays.
        - set_state(self, state): Sets the current state of the game.
        - get_action_table(self): Returns the current action table.
        - get_piece_to_move(self, state, action): Retrieves the piece to move for a given state and action.
        - reset(self): Resets the action table and piece-to-move table to np.nan.
        - update_action_table(self, action, piece, value): Updates the action table and piece-to-move table with new values if the current value is np.nan.
3. **Q-Learning Implementation**:
    - **Initialization**:
        - Define the state and action space based on the Ludo game.
        - Initialize the Q-table with zeros to represent unlearned values.
    - **Training Loop**:
        - For each episode (game simulation):
            - Initialize the starting state.
            - For each step in the episode:

        - Select an action using an epsilon-greedy policy to balance exploration and exploitation.
        - Execute the action and observe the new state and reward.
        - Update the Q-table using the Q-learning update rule: $Q(s,a)=Q(s,a)+\alpha[r+\gamma \max_{a'} Q(s',a')-Q(s,a)]$ Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$Q(s,a)=Q(s,a)+\alpha[r+\gamma a' \max Q(s',a')-Q(s,a)]$ where $\alpha$ is the learning rate and $\gamma$ is the discount factor.
        - Transition to the new state.
    - **Update Rule**:
        - The Q-table is updated based on the received reward and the maximum expected future reward.

**Highlights**

- **Q-Learning Algorithm**: Utilizes Q-learning to enable the AI to learn optimal strategies by updating the Q-table based on rewards.
- **State-Action Representation**: Efficiently represents state-action pairs using numpy arrays for quick updates and retrieval.
- **Epsilon-Greedy Policy**: Balances exploration (trying new actions) and exploitation (using known rewarding actions) to optimize learning.
- **Dynamic Learning**: The AI agent continuously improves its strategy by updating the Q-table based on the rewards received from different actions.
- **Modular Code**: The code is structured with clear classes and methods, making it easy to extend or modify for further improvements.

**Conclusion**

The project successfully demonstrates the application of Q-learning to the game of Ludo. Through iterative training and updating of the Q-table, the AI agent learns to make better decisions and improve its gameplay strategy. The use of numpy ensures efficient handling of the state-action values, and the epsilon-greedy policy provides a balanced approach to exploration and exploitation. This project lays the groundwork for further enhancements, such as more sophisticated reward structures, different exploration strategies, or adapting the algorithm to other board games. The results show that reinforcement learning can be effectively used to develop intelligent game-playing agents.