Name : Tavhare Ruchita Sharad

class : BE AI&DS

Roll No.: 61

Subject: Computer Laboratory-I(ML)

Title:Build a Tic-Tac-Toe game using reinforcement learning in Python by using following tasks

a. Setting up the environment

b. Defining the Tic-Tac-Toe game

c. Building the reinforcement learning model

d. Training the model

e. Testing the model

In [27]:
```
import numpy as np
```

In [37]:
```
class TicTacToeEnvironment:
    def __init__(self):
        self.state = [0] * 9
        self.is_terminal = False

    def reset(self):
        self.state = [0] * 9
        self.is_terminal = False

    def get_available_moves(self):
        return [i for i, mark in enumerate(self.state) if mark == 0]

    def make_move(self, move, player_mark):
        self.state[move] = player_mark

    def check_win(self, player_mark):
        winning_combinations = [
            [0, 1, 2], [3, 4, 5], [6, 7, 8],
            [0, 3, 6], [1, 4, 7], [2, 5, 8],
            [0, 4, 8], [2, 4, 6]
        ]
        for combo in winning_combinations:
            if all(self.state[i] == player_mark for i in combo):
                self.is_terminal = True
                return True
        return False

    def is_draw(self):
        # A draw occurs when no empty cells remain and no one has won
        if 0 not in self.state and not self.is_terminal:
            self.is_terminal = True
            return True
        return False
```

In [39]:
```
class QLearningAgent:
    def __init__(self, learning_rate=0.9, discount_factor=0.9, exploration_rat
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate
        self.q_table = np.zeros((3**9, 9))  # 3^9 possible states, 9 possible

    def get_state_index(self, state):
        state_index = 0
        for i, mark in enumerate(state):
            state_index += (3 ** i) * (mark + 1)
        return state_index

    def choose_action(self, state, available_moves):
        state_index = self.get_state_index(state)
        if np.random.random() < self.exploration_rate:
            return np.random.choice(available_moves)  # Exploration
        else:
            # Exploitation: choose the best Q-value among available moves
            q_values = self.q_table[state_index, available_moves]
            return available_moves[np.argmax(q_values)]

    def update_q_table(self, state, action, next_state, reward):
        state_index = self.get_state_index(state)
        next_state_index = self.get_state_index(next_state) if next_state is n
        max_q_value = np.max(self.q_table[next_state_index]) if next_state is

        # Q-learning update rule
        self.q_table[state_index, action] = (
            (1 - self.learning_rate) * self.q_table[state_index, action]
            + self.learning_rate * (reward + self.discount_factor * max_q_valu
        )
```

In [41]:
```
def evaluate_agents(agent1, agent2, num_episodes=1000):
    environment = TicTacToeEnvironment()
    agent1_wins = 0
    agent2_wins = 0
    draws = 0

    for _ in range(num_episodes):
        environment.reset()
        current_agent = agent1

        while not environment.is_terminal:
            available_moves = environment.get_available_moves()
            current_state = environment.state.copy()
            action = current_agent.choose_action(current_state, available_move

            # Player mark: +1 for agent1, -1 for agent2
            player_mark = 1 if current_agent == agent1 else -1
            environment.make_move(action, player_mark)

            # Check win
            if environment.check_win(player_mark):
                current_agent.update_q_table(current_state, action, None, 10)
                if current_agent == agent1:
                    agent1_wins += 1
                else:
                    agent2_wins += 1
                break

            # Check draw
            elif environment.is_draw():
                current_agent.update_q_table(current_state, action, None, 0)
                draws += 1
                break

            # If the game continues
            next_state = environment.state.copy()

            # Small negative reward for a non-terminal move to encourage faste
            current_agent.update_q_table(current_state, action, next_state, -0

            # Switch player (this is the corrected line)
            current_agent = agent2 if current_agent == agent1 else agent1

    return agent1_wins, agent2_wins, draws
```

In [43]:
```
# Create agents
agent1 = QLearningAgent()
agent2 = QLearningAgent()
# Evaluate agents
agent1_wins, agent2_wins, draws = evaluate_agents(agent1, agent2)
# Print results
print(f"Agent 1 wins: {agent1_wins}")
print(f"Agent 2 wins: {agent2_wins}")
print(f"Draws: {draws}")
```

```
Agent 1 wins: 617
Agent 2 wins: 268
Draws: 115
```

In [ ]: