CL1-06

July 24, 2025

[ ]:
```
'''NAME:Aher Swami Sandip
ROLL NO.01
COURSE: AI&DS
CLASS: BE
SUB:Computer Laboratory-I (Machine Learning)'''
```

[ ]:
```
'''
PRACTICAL NO-06:
     Reinforcement Learning:
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
'''
```

[1]:
```
import numpy as np
import random
```

[3]:
```
# The Tic-Tac-Toe Environment
class TicTacToe:
    def __init__(self):
        self.board = np.zeros((3, 3), dtype=int)
        self.done = False
        self.winner = None

    def reset(self):
        self.board = np.zeros((3, 3), dtype=int)
        self.done = False
        self.winner = None
        return self.board

    def available_actions(self):
        return list(zip(*np.where(self.board == 0)))
```

```
    def step(self, action, player):
        if self.board[action] != 0:
            raise ValueError("Invalid Action")
        self.board[action] = player
        self.done, self.winner = self.check_winner()
        reward = 0
        if self.done:
            if self.winner == player:
                reward = 1  # Winning
            elif self.winner == 0:
                reward = 0.5  # Draw
            else:
                reward = -1  # Losing
        return self.board, reward, self.done

    def check_winner(self):
        for i in range(3):
            if np.all(self.board[i, :] == self.board[i, 0]) and self.board[i,␣
↪0] != 0:
                return True, self.board[i, 0]
            if np.all(self.board[:, i] == self.board[0, i]) and self.board[0,␣
↪i] != 0:
                return True, self.board[0, i]

        if np.all(np.diag(self.board) == self.board[0, 0]) and self.board[0, 0]␣
↪!= 0:
            return True, self.board[0, 0]
        if np.all(np.diag(np.fliplr(self.board)) == self.board[0, 2]) and self.
↪board[0, 2] != 0:
            return True, self.board[0, 2]

        if np.all(self.board != 0):
            return True, 0  # Draw

        return False, None
```

[5]:
```
# Define the Q-Learning Agent
class QLearningAgent:
    def __init__(self, epsilon=0.1, alpha=0.5, gamma=0.9):
        self.q_table = {}
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma

    def get_q_value(self, state, action):
        return self.q_table.get((state, action), 0.0)
```

```
    def update_q_value(self, state, action, reward, next_state,␣
↪available_actions):
        best_next_action = max(self.q_table.get((next_state, a), 0.0) for a in␣
↪available_actions)
        self.q_table[(state, action)] = (1 - self.alpha) * self.
↪get_q_value(state, action) + self.alpha * (reward + self.gamma *␣
↪best_next_action)

    def choose_action(self, state, actions):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(actions)
        q_values = [self.get_q_value(state, a) for a in actions]
        max_q = max(q_values)
        return actions[q_values.index(max_q)]
```

[7]:
```
# The Training Function
def train(agent, episodes=10000):
    for episode in range(episodes):
        game = TicTacToe()
        state = game.reset()
        player = 1
        while True:
            available_actions = game.available_actions()
            action = agent.choose_action(state.tobytes(), available_actions)
            next_state, reward, done = game.step(action, player)
            if done:
                agent.update_q_value(state.tobytes(), action, reward,␣
↪next_state.tobytes(), available_actions)
                break
            agent.update_q_value(state.tobytes(), action, 0, next_state.
↪tobytes(), game.available_actions())
            state = next_state
            player = 2 if player == 1 else 1
```

[9]:
```
# Train the Agent
agent = QLearningAgent()
train(agent)
```

[10]:
```
# Test the Model by Playing a Game
def play_game(agent):
    game = TicTacToe()
    state = game.reset()
    player = 1
    while True:
        print(game.board)
        if player == 1:
            available_actions = game.available_actions()
```

```
            action = agent.choose_action(state.tobytes(), available_actions)
        else:
            action = random.choice(game.available_actions())  # Random Agent
        next_state, reward, done = game.step(action, player)
        if done:
            print("Game over!")
            print(game.board)
            if game.winner == 0:
                print("It's a draw!")
            else:
                print(f"Player {game.winner} wins!")
            break
        state = next_state
        player = 2 if player == 1 else 1

play_game(agent)
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
[[1 0 0]
 [0 0 0]
 [0 0 0]]
[[1 2 0]
 [0 0 0]
 [0 0 0]]
[[1 2 1]
 [0 0 0]
 [0 0 0]]
[[1 2 1]
 [0 0 0]
 [2 0 0]]
[[1 2 1]
 [1 0 0]
 [2 0 0]]
[[1 2 1]
 [1 0 0]
 [2 0 2]]
[[1 2 1]
 [1 1 0]
 [2 0 2]]
[[1 2 1]
 [1 1 2]
 [2 0 2]]
Game over!
[[1 2 1]
 [1 1 2]
 [2 1 2]]
```

```
It's a draw!
```

[ ]: