EXPERIMENT NO. 6 C

- **Aim**: 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

- **Hardware Requirement:**

- 6 GB free disk space.
- 2 GB RAM.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).
- Virtualization is available with the KVM hypervisor
- Intel 64 and AMD64 architectures

- **Software Requirement:**
  Jypiter Nootbook/Ubuntu

- **Theory:**

In reinforcement learning, developers devise a method of rewarding desired behaviors and punishing negative behaviors. This method assigns positive values to the desired actions to encourage the agent and negative values to undesired behaviors. This programs the agent to seek long-term and maximum overall reward to achieve an optimal solution.

These long-term goals help prevent the agent from stalling on lesser goals. With time, the agent learns to avoid the negative and seek the positive. This learning method has been adopted in artificial intelligence (AI) as a way of directing unsupervised machine learning through rewards and penalties.

Common reinforcement learning algorithms

Rather than referring to a specific algorithm, the field of reinforcement learning is made up of several algorithms that take somewhat different approaches. The differences are mainly due to their strategies for exploring their environments.

- State-action-reward-state-action (SARSA). This reinforcement learning algorithm starts by giving the

agent what's known as a *policy*. The policy is essentially a probability that tells it the odds of certain actions resulting in rewards, or beneficial states.

- Q-learning. This approach to reinforcement learning takes the opposite approach. The agent receives no policy, meaning its exploration of its environment is more self-directed.
- Deep Q-Networks. These algorithms utilize neural networks in addition to reinforcement learning techniques. They utilize the self-directed environment exploration of reinforcement learning. Future actions are based on a random sample of past beneficial actions learned by the neural network.

Implementation:

```
import numpy as np

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_states = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
        [0, 4, 8], [2, 4, 6]  # diagonals
        ]
        for state_indices in winning_states:
        if all(self.state[i] == player_mark for i in state_indices):
            self.is_terminal = True
            return True
        return False
```

```
    def is_draw(self):
    return 0 not in self.state

class QLearningAgent:
    def __init__(self, learning_rate=0.9, discount_factor=0.9, exploration_rate=0.3):
    self.learning_rate = learning_rate
    self.discount_factor = discount_factor
    self.exploration_rate = exploration_rate
    self.q_table = np.zeros((3**9, 9))

    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)
    else:
    return np.argmax(self.q_table[state_index, available_moves])

    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 not None else None
  max_q_value = np.max(self.q_table[next_state_index]) if next_state is not None else 0
    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_value)

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_moves)
        environment.make_move(action, 1 if current_agent == agent1 else -1)

        if environment.check_win(1 if current_agent == agent1 else -1):
            current_agent.update_q_table(current_state, action, None, 10)
            if current_agent == agent1:
                agent1_wins += 1
            else:
                agent2_wins += 1
            break
        elif environment.is_draw():
            current_agent.update_q_table(current_state, action, None, 0)
            draws += 1
            break

        next_state = environment.state.copy()
        reward = 0
        if environment.check_win(1 if current_agent == agent1 else -1):
            reward = -10
        current_agent.update_q_table(current_state, action, next_state, reward)

        current_agent = agent2 if current_agent == agent1 else agent1

    return agent1_wins, agent2_wins, draws

# 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: 458
Agent 2 wins: 470
Draws: 72
```

TicTacToeEnvironment:

This class represents the Tic-Tac-Toe game environment. It maintains the current state of the game, checks for a win or draw, and provides methods to reset the game and make moves.

The __init__ method initializes the game state and sets the terminal flag to False.

The reset method resets the game state and the terminal flag.

The get_available_moves method returns a list of indices representing the available moves in the current game state.

The make_move method updates the game state by placing a player's mark at the specified move index.

The check_win method checks if a player has won the game by examining the current state.

The is_draw method checks if the game has ended in a draw.

QLearningAgent:

This class represents the Q-learning agent. It learns to play Tic-Tac-Toe by updating a Q-table based on the rewards received during gameplay.

The __init__ method initializes the learning rate, discount factor, exploration rate, and the Q-table.

The get_state_index method converts the current game state into a unique index for indexing the Q-table.

The choose_action method selects the action (move) to be taken based on the current game state and the exploration-exploitation tradeoff using the epsilon-greedy policy.

The update_q_table method updates the Q-table based on the current state, action, next state, and the reward received.

evaluate_agents:

This function performs the evaluation of two Q-learning agents by playing multiple episodes of Tic-Tac-Toe games.

It takes the two agents and the number of episodes to play as input.

In each episode, the environment is reset, and the agents take turns making moves until the game is over (either a win or a draw).

The agents update their Q-tables based on the rewards received during the episode.

The function keeps track of the wins and draws for each agent and returns the counts.

Main code:

The main code creates two Q-learning agents, agent1 and agent2, using the QLearningAgent class.

The evaluate_agents function is called to evaluate the agents by playing a specified number of episodes.

The results (number of wins and draws) for each agent are printed.

The Q-learning algorithm involves the following steps:

The agents choose their moves based on the current game state and the exploration-exploitation policy.

The environment updates the game state based on the chosen moves.

The environment checks if the game has ended (win or draw).

The agents update their Q-tables based on the rewards received.

The agents continue playing until the specified number of episodes is completed.