Tic-Tac-Toe Game with Reinforcement Learning

In [ ]:

```
import tensorflow as tf
import numpy as np
import random
```

In [ ]:

```
class TicTacToe:
    """
    A class representing a simple Tic-Tac-Toe game.

    Attributes:
        board (numpy.ndarray): A 3x3 numpy array representing the game board.
        Each cell can be empty (0), marked by 'X' (1), or marked by 'O' (2).
        players (list): A list containing the symbols used by the players ('X' and 'O').
        current_player (str): The symbol of the player whose turn it currently is.
        winner (str): The symbol of the player who has won the game (if any).
        game_over (bool): A boolean indicating whether the game has ended.

    Methods:
        __init__(): Initialize a new Tic-Tac-Toe game with an empty board.
        reset(): Reset the game to its initial state.
        available_moves(): Get a list of available (empty) positions on the board.
        make_move(move): Make a move on the board if it's a valid and available move.
        switch_player(): Switch the current player to the other player.
        check_winner(): Check if there is a winner and update the `winner` and `game_ove
        print_board(): Print the current state of the game board.

    Example Usage:
        game = TicTacToe()
        game.make_move((0, 0))
        game.make_move((1, 1))
        game.print_board()
        if game.winner:
            print(f"The winner is: {game.winner}")
        elif game.game_over:
            print("It's a draw!")
    """
    def __init__(self):
        """
        Initialize a new Tic-Tac-Toe game.

        The game starts with an empty 3x3 board, and 'X' always goes first.
        """
        self.board = np.zeros((3, 3))
        self.players = ['X', 'O']
        self.current_player = None
        self.winner = None
        self.game_over = False

    def reset(self):
        """
        Reset the game to its initial state.

        This method clears the board, resets the current player, and sets the game statu
        """
        self.board = np.zeros((3, 3))
        self.current_player = None
        self.winner = None
        self.game_over = False

    def available_moves(self):
         """
        Get a list of available (empty) positions on the board.

        Returns:
```

```
            list: A list of tuples representing available positions as (row, column).
        """
        moves = []
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    moves.append((i, j))
        return moves

    def make_move(self, move):
        """
        Make a move on the board if it's a valid and available move.

        Args:
            move (tuple): A tuple representing the target position as (row, column).

        Returns:
            bool: True if the move was successfully made, False otherwise.
        """
        if self.board[move[0]][move[1]] != 0:
            return False
        self.board[move[0]][move[1]] = self.players.index(self.current_player) + 1
        self.check_winner()
        self.switch_player()
        return True

    def switch_player(self):
        """
        Switch the current player to the other player.
        """
        if self.current_player == self.players[0]:
            self.current_player = self.players[1]
        else:
            self.current_player = self.players[0]

    def check_winner(self):
        """
        Check if there is a winner and update the `winner` and `game_over` attributes.
        """
        # Check rows
        for i in range(3):
            if self.board[i][0] == self.board[i][1] == self.board[i][2] != 0:
                self.winner = self.players[int(self.board[i][0] - 1)]
                self.game_over = True
        # Check columns
        for j in range(3):
            if self.board[0][j] == self.board[1][j] == self.board[2][j] != 0:
                self.winner = self.players[int(self.board[0][j] - 1)]
                self.game_over = True
        # Check diagonals
        if self.board[0][0] == self.board[1][1] == self.board[2][2] != 0:
            self.winner = self.players[int(self.board[0][0] - 1)]
            self.game_over = True
        if self.board[0][2] == self.board[1][1] == self.board[2][0] != 0:
            self.winner = self.players[int(self.board[0][2] - 1)]
            self.game_over = True

    def print_board(self):
        """
        Print the current state of the game board.
        """
```

```
        print("-------------")
        for i in range(3):
            print("|", end=' ')
            for j in range(3):
                print(self.players[int(self.board[i][j] - 1)] if self.board[i][j] != 0 e
            print()
            print("-------------")
```

In [ ]:

```
# Create a TicTacToe game instance
game = TicTacToe()

# Set the current player to 'X' (Player 1)
game.current_player = game.players[0]

# Print the initial empty game board
game.print_board()

# Start the game loop until it's over
while not game.game_over:

    # Prompt the current player for their move (row and column)
    move = input(f"{game.current_player}'s turn. Enter row and column (e.g. 0 0): ")
    move = tuple(map(int, move.split()))
    while move not in game.available_moves():
        move = input("Invalid move. Try again: ")
        move = tuple(map(int, move.split()))

    # Make the valid move on the game board
    game.make_move(move)
    game.print_board()

# Check if there is a winner or if it's a tie
if game.winner:
    print(f"{game.winner} wins!")
else:
    print("It's a tie!")
```

In [ ]:

```
import random

class QLearningAgent:
    """
    A class representing a Q-Learning agent for a reinforcement learning task.

    Attributes:
        Q (dict): A dictionary to store Q-values for state-action pairs.
        alpha (float): The learning rate, controlling the impact of new information on Q
        epsilon (float): The exploration rate, controlling the probability of taking a r
        discount_factor (float): The discount factor for future rewards.

    Methods:
        __init__(self, alpha, epsilon, discount_factor): Initialize a Q-Learning agent w
        get_Q_value(self, state, action): Get the Q-value for a specific state-action pa
        choose_action(self, state, available_moves): Choose an action based on Q-values
        update_Q_value(self, state, action, reward, next_state): Update the Q-value for

    Example Usage:
        agent = QLearningAgent(alpha=0.1, epsilon=0.2, discount_factor=0.9)
        state = current_game_state()
        available_moves = get_available_moves(state)

        # Choose an action based on Q-values and exploration rate
        action = agent.choose_action(state, available_moves)

        # Update Q-value based on received reward and the next state
        agent.update_Q_value(state, action, reward, next_state)
    """

    def __init__(self, alpha, epsilon, discount_factor):
        """
        Initialize a Q-Learning agent.

        Args:
            alpha (float): The learning rate, controlling the impact of new information
            epsilon (float): The exploration rate, controlling the probability of taking
            discount_factor (float): The discount factor for future rewards.
        """
        self.Q = {}
        self.alpha = alpha
        self.epsilon = epsilon
        self.discount_factor = discount_factor

    def get_Q_value(self, state, action):
        """
        Get the Q-value for a specific state-action pair.

        Args:
            state (hashable): The current state.
            action (hashable): The action taken in the current state.

        Returns:
            float: The Q-value for the given state-action pair.
        """
        if (state, action) not in self.Q:
            self.Q[(state, action)] = 0.0
        return self.Q[(state, action)]
```

```
    def choose_action(self, state, available_moves):
        """
        Choose an action based on Q-values and exploration rate.

        Args:
            state (hashable): The current state.
            available_moves (list): A list of available actions in the current state.

        Returns:
            hashable: The chosen action.
        """
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(available_moves)
        else:
            Q_values = [self.get_Q_value(state, action) for action in available_moves]
            max_Q = max(Q_values)
            if Q_values.count(max_Q) > 1:
                best_moves = [i for i in range(len(available_moves)) if Q_values[i] == m
                i = random.choice(best_moves)
            else:
                i = Q_values.index(max_Q)
            return available_moves[i]

    def update_Q_value(self, state, action, reward, next_state):
        """
        Update the Q-value for a state-action pair.

        Args:
            state (hashable): The current state.
            action (hashable): The action taken in the current state.
            reward (float): The reward received for taking the action.
            next_state (hashable): The resulting state after taking the action.
        """
        next_Q_values = [self.get_Q_value(next_state, next_action) for next_action in Ti
        max_next_Q = max(next_Q_values) if next_Q_values else 0.0
        self.Q[(state, action)] += self.alpha * (reward + self.discount_factor * max_ne
```

In [ ]:

```
def train(num_episodes, alpha, epsilon, discount_factor):
     """
    Train a Q-Learning agent by playing Tic-Tac-Toe episodes.

    Args:
        num_episodes (int): The number of episodes (games) to play for training.
        alpha (float): The learning rate for updating Q-values.
        epsilon (float): The exploration rate, controlling random actions during trainir
        discount_factor (float): The discount factor for future rewards.

    Returns:
        QLearningAgent: A trained Q-Learning agent.
    """
    agent = QLearningAgent(alpha, epsilon, discount_factor)
    for i in range(num_episodes):
        state = TicTacToe().board
        while not TicTacToe(state).game_over():
            available_moves = TicTacToe(state).available_moves()
            action = agent.choose_action(state, available_moves)
            next_state, reward = TicTacToe(state).make_move(action)
            agent.update_Q_value(state, action, reward, next_state)
            state = next_state
    return agent
```

In [ ]:

```
def test(agent, num_games):
    """
    Test a trained Q-Learning agent by playing Tic-Tac-Toe games.

    Args:
        agent (QLearningAgent): A trained Q-Learning agent.
        num_games (int): The number of games to play for testing.

    Returns:
        float: The win percentage of the agent in the test games.
    """
    num_wins = 0
    for i in range(num_games):
        state = TicTacToe().board
        while not TicTacToe(state).game_over():
            if TicTacToe(state).player == 1:
                action = agent.choose_action(state, TicTacToe(state).available_moves())
            else:
                action = random.choice(TicTacToe(state).available_moves())
            state, reward = TicTacToe(state).make_move(action)
        if reward == 1:
            num_wins += 1
    return num_wins / num_games * 100
```

In [ ]:

```
# Train the Q-learning agent
agent = train(num_episodes=100000, alpha=0.5, epsilon=0.1, discount_factor=1.0)

# Test the Q-learning agent
win_percentage = test(agent, num_games=1000)
print("Win percentage: {:.2f}%".format(win_percentage))
```

In [ ]:

```

```