CSE 584, HOMEWORK 2

Name: Yogeshvar Reddy Kallam IdNum: 920385794 email: yvk5381@psu.edu

Abstract

This code basically implements a Q-learning reinforcement learning on classic Tic-Tac-Toe; the aim is to train an AI agent to iteratively learn the best way to play against it. This code presents a Q-learning framework-a model-free reinforcement learning method-to develop an improving AI which would be able to play Tic-Tac-Toe with an increasingly higher level of skill. It does so by iteratively improving the estimates of the Q-values-or values of state-action pairs-through repeated interactions with the game environment for making more informative decisions in subsequent games.

The agent starts having no prior knowledge of playing the game effectively. The agent explores various actions and results in terms of rewards-for example, whether he wins, loses, or draws a game. Over time, AI balances exploration-trying new moves randomly to gain experience-with exploitation-choosing moves known to be effective based on learned Q-values. Epsilon-greedy policy maintains this balance-a high epsilon value provides exploration, while for a low epsilon value, the agent shifts towards exploitation with gathered experience.

This Tic-Tac-Toe game here has been visualized in a grid where an agent playing "X" competes with either an opponent randomly or with a human player. The state of the game is thereafter defined as a list of nine positions, each having a value that could either be an empty square, the agent's move-"X", or the opponent's move-"O". The Q-table is actually the heart of the learning process, which is nothing but a table where for every possible game state-action pair, a Q-value is assigned, which estimates the expected reward-to-go from that action in that state.

The agent selects an action in each game, according to its current Q-table and the policy of epsilon-greedy. Then, based on the outcome, the agent will receive a reward-for instance, a reward of +1 for winning, 0.5 for drawing, and -1 for losing-and it will be updated using the Bellman equation:

Q(s,a)=Q(s,a)+α(r+γmaxQ(s′,a′)−Q(s,a))Q(s, a) = Q(s, a) + \alpha \left(r + \gamma \max Q(s', a') - Q(s, a) \right)Q(s,a)=Q(s,a)+α(r+γmaxQ(s′,a′)−Q(s,a))

where:

- Q(s,a)Q(s, a)Q(s,a) is the current Q-value for state sss and action aaa,
- α\alphaα is the learning rate, controlling how much new information overrides old knowledge,

- rrr is the immediate reward received after taking action aaa,
- γ\gammaγ is the discount factor, controlling how much future rewards are valued compared to immediate ones,
- maxQ(s′,a′)\max Q(s', a')maxQ(s′,a′) is the maximum future reward for the next state s′s's′.

The Q-learning algorithm is off-policy; it updates its Q-values concerning the best possible future actions, not necessarily the actions the agent actually took. That will grant the agent to converge to an optimal strategy over time and therefore potentially make it unbeatable in Tic-Tac-Toe.

Process Overview:

The environment is initialized: The board for playing Tic-Tac-Toe is created and the Q-table is initialized to zeros, with each cell in the Q-table holding an estimate of a value for the expected reward regarding a particular state-action combination.
Agent Training: The agent plays numerous episodes of games while learning by interacting with the environment-that is, making moves on the board. It updates its Q-values after every move, taking into account the reward received and an estimate of the future rewards.
Exploration and Exploitation: The agent will follow the epsilon-greedy policy through training-the choice between exploring new actions by making random moves or exploiting known good actions by moving using the highest Q-value. Epsilon starts high to encourage exploration and decays as the process goes on, leading the agent to start exploiting more when it gets confident in its strategy.
Testing and Evaluation: Once trained, the agent is allowed to play against a human or another random opponent to see how well it performs. This agent will be increasingly difficult to defeat as time progresses because the Q-values are converging to the optimal values; it uses its learned policy to anticipate the best possible moves.
Learning Dynamics: The code uses hyperparameters for learning rate (α) and discount factor (γ), controlling the trade-off between valuing immediate rewards versus future rewards. It is important to find a proper balance between these two parameters, because it affects the performance of the agent.

Code Section with Line-by-Line Comments

```
import numpy as np
import random

# Define the Tic-Tac-Toe game environment
class TicTacToe:
    def __init__(self):
        # Initialize a 3x3 board represented as a 1D list with empty spaces
        self.board = [' '] * 9
        self.current_winner = None  # Track the winner (None if no winner yet)

    # Reset the board to start a new game
    def reset(self):
        self.board = [' '] * 9  # Reset board to empty state
        self.current_winner = None  # Reset the winner
        return self.get_state()  # Return the initial state of the board

    # Get the available moves (empty spots on the board)
    def available_moves(self):
        return [i for i, x in enumerate(self.board) if x == ' ']  # Return indices where the board is
empty

    # Check if the board has empty squares left
    def empty_squares(self):
        return ' ' in self.board  # Return True if there are empty spots

    # Place a move on the board
    def make_move(self, square, player):
        if self.board[square] == ' ':
            self.board[square] = player  # Mark the board with 'X' or 'O'
            # Check if the move wins the game
            if self.winner(square, player):
                self.current_winner = player  # Mark the player as the winner
            return True
        return False  # Return False if the spot is already taken

    # Check if the player has won the game
    def winner(self, square, player):
        # Define winning conditions: rows, columns, diagonals
        win_conditions = [
            [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 condition in win_conditions:
            if all(self.board[i] == player for i in condition):  # Check if all spots in a win condition are
filled by the player
                return True
        return False  # Return False if no winning condition is met

    # Return the current state of the board as a tuple (to be used for Q-table indexing)
    def get_state(self):
        return tuple(self.board)

    # Display the current state of the board
    def render(self):
        # Print the board in a human-readable format (3x3 grid)
        print('\n'.join([' | '.join(self.board[i:i+3]) for i in range(0, 9, 3)]))

# Define the Q-Learning agent
class QLearningAgent:
    def __init__(self, epsilon=0.1, alpha=0.5, gamma=0.9):
        self.q_table = {}  # Initialize an empty Q-table (dictionary)
        self.epsilon = epsilon  # Exploration rate (probability of choosing a random action)
        self.alpha = alpha  # Learning rate (controls how much new information is used)
        self.gamma = gamma  # Discount factor (how much to value future rewards)

    # Choose an action based on the epsilon-greedy policy
    def choose_action(self, state, available_moves):
        if random.uniform(0, 1) < self.epsilon:  # With probability epsilon, explore
            return random.choice(available_moves)  # Choose a random move from available
moves
        else:
            # Exploit: Select the move with the highest Q-value
            q_values = [self.q_table.get((state, move), 0) for move in available_moves]  # Get
Q-values for all available moves
            max_q_value = max(q_values)  # Find the max Q-value
            return available_moves[q_values.index(max_q_value)]  # Return the action with the
highest Q-value

    # Update the Q-value using the Q-learning update rule
    def learn(self, state, action, reward, next_state, done, available_moves):
        old_q_value = self.q_table.get((state, action), 0)  # Get the old Q-value (0 if not available in
Q-table)
        if done:
            # If the game ends (win, lose, or draw), there is no future state
            new_q_value = reward  # The new Q-value is simply the reward
        else:
            # If the game continues, estimate the future reward using the next state's best action
            next_q_values = [self.q_table.get((next_state, move), 0) for move in available_moves]
            new_q_value = reward + self.gamma * max(next_q_values)  # Q-learning formula
        # Update the Q-value in the Q-table with the new Q-value
        self.q_table[(state, action)] = old_q_value + self.alpha * (new_q_value - old_q_value)

# Training loop to train the Q-learning agent
def train_agent(episodes=5000):
    agent = QLearningAgent(epsilon=0.2, alpha=0.5, gamma=0.9)  # Initialize the Q-learning
agent
    env = TicTacToe()  # Initialize the Tic-Tac-Toe environment
    # Loop through a specified number of episodes (games)
    for episode in range(episodes):
        state = env.reset()  # Reset the environment for each new game
        done = False  # Track if the game is done (win or tie)

        # Play until the game ends
        while not done:
            available_moves = env.available_moves()  # Get available moves
            action = agent.choose_action(state, available_moves)  # Choose an action using
epsilon-greedy policy
            env.make_move(action, 'X')  # Assume the agent is player 'X'

            if env.current_winner:
                reward = 1  # Reward for winning
                done = True  # End the game if agent wins
            elif not env.empty_squares():
                reward = 0.5  # Reward for a tie
                done = True  # End the game if it's a tie
            else:
                reward = 0  # No reward for intermediate moves
                # Simulate a random opponent ('O') move
                if env.empty_squares():
                    opponent_move = random.choice(env.available_moves())
                    env.make_move(opponent_move, 'O')  # 'O' plays randomly
                    if env.current_winner:
                        reward = -1  # Negative reward if agent loses
                        done = True  # End the game if opponent wins
                    elif not env.empty_squares():
                        reward = 0.5  # Reward for a tie if no moves left
                        done = True

            next_state = env.get_state()  # Get the new state after the move
            agent.learn(state, action, reward, next_state, done, env.available_moves())  # Update
the Q-values
            state = next_state  # Move to the next state

    return agent  # Return the trained agent after training

# Function to play against the trained agent
def play_game(agent):
    env = TicTacToe()  # Initialize a new game
    state = env.reset()  # Reset the board for the start of the game

    # Play until the board is full or someone wins
    while env.empty_squares():
        env.render()  # Display the current state of the board
        available_moves = env.available_moves()  # Get available moves

        # Human player ('O') move
        human_move = int(input(f"Choose your move (0-8): "))  # Get input from the human player
        env.make_move(human_move, 'O')  # Make the human's move

        if env.current_winner:
            print("You won!")  # Display message if human wins
            break

        # Agent's move ('X')
        action = agent.choose_action(state, available_moves)  # Let the agent choose its move
        env.make_move(action, 'X')  # Make the agent's move

        if env.current_winner:
            print("Agent won!")  # Display message if agent wins
            break
        elif not env.empty_squares():
            print("It's a tie!")  # Display message if it's a tie
            break

    env.render()  # Display the final state of the board

# Train the agent
trained_agent = train_agent(episodes=10000)

# Play against the trained agent
play_game(trained_agent)
```

Explanation:

The environment is initialized: The board for playing Tic-Tac-Toe is created and the Q-table is initialized to zeros, with each cell in the Q-table holding an estimate of a value for the expected reward regarding a particular state-action combination.

Agent Training: The agent plays numerous episodes of games while learning by interacting with the environment-that is, making moves on the board. It updates its Q-values after every move, taking into account the reward received and an estimate of the future rewards.

Exploration and Exploitation: The agent will follow the epsilon-greedy policy through training-the choice between exploring new actions by making random moves or exploiting known good actions by moving using the highest Q-value. Epsilon starts high to encourage exploration and decays as the process goes on, leading the agent to start exploiting more when it gets confident in its strategy.

Testing and Evaluation: Once trained, the agent is allowed to play against a human or another random opponent to see how well it performs. This agent will be increasingly difficult to defeat as time progresses because the Q-values are converging to the optimal values; it uses its learned policy to anticipate the best possible moves.

Learning Dynamics: The code uses hyperparameters for learning rate (α) and discount factor (γ), controlling the trade-off between valuing immediate rewards versus future rewards. It is important to find a proper balance between these two parameters, because it affects the performance of the agent.

References

https://github.com/swetha0404/Reinforcement_Q-Learning
https://github.com/khpeek/Q-learning-Tic-Tac-Toe