Q-LEARNING IN TIC-TAC-TOE: AN AI AGENT FOR OPTIMAL GAMEPLAY

This Program uses q-learning to implement the Tic Tac Toe game

Tic-Tac-Toe

Contents

Abstract

This project explores the application of Q-learning, a reinforcement learning algorithm, to the game of Tic-Tac-Toe. The goal is to develop an AI agent capable of learning optimal strategies through self-play, thereby maximizing its chances of winning against a random opponent. Q-learning, based on the principles of trial and error, enables the agent to iteratively improve its decision-making by updating a Q-table, which represents the expected utility of taking specific actions from various game states.

The agent's learning process involved exploring the game environment through an epsilon-greedy strategy, balancing between exploration of new moves and exploitation of known beneficial actions. By training over multiple episodes, the agent progressively improved, achieving a high win rate by the end of training. The Q-table was also saved and reloaded using Python's pickle library, facilitating extended learning over multiple training sessions. Results indicate that the agent successfully learned optimal moves, adapting to opponent strategies and avoiding losing states.

This study highlights the efficacy of Q-learning in simple game environments, with Tic-Tac-Toe serving as a foundational model for understanding reinforcement learning principles. Future improvements could involve extending this approach to more complex games or employing neural networks for enhanced decision-making capabilities.

Introduction

The development of intelligent game-playing agents has been a cornerstone in the study of artificial intelligence and machine learning, as it provides a structured environment for testing learning algorithms and decision-making processes. Tic-Tac-Toe, a simple yet strategically rich game, serves as an ideal platform for exploring the fundamentals of reinforcement learning. The objective of this project is to implement a Q-learning algorithm to train an AI agent that can play Tic-Tac-Toe optimally, learning effective strategies entirely through self-play.

Reinforcement learning (RL) is a subset of machine learning in which an agent learns by interacting with its environment and receiving feedback in the form of rewards or penalties. Among RL algorithms, Q-learning is particularly effective for environments with defined states and actions, like games. Q-learning allows an agent to learn a policy that maximizes its cumulative reward by adjusting a Q-table, a data structure that records expected rewards for state-action pairs. This approach is well-suited to Tic-Tac-Toe, where every board configuration (state) leads to a finite set of possible moves (actions).

The motivation behind this project is to understand and demonstrate how Q-learning can guide an AI agent toward optimal decision-making without prior knowledge of the game rules. By using an epsilon-greedy strategy, the agent balances exploration of new moves with exploitation of moves known to be beneficial. Over multiple training episodes, the agent refines its Q-table, learning to avoid losing states and prioritize winning moves.

Background

1. Introduction to Reinforcement Learning:

- Reinforcement learning (RL) is a subset of machine learning where an agent learns to make decisions by taking actions in an environment to maximize cumulative reward.
- In RL, the agent interacts with the environment by exploring and exploiting actions based on its current knowledge. This interaction helps the agent learn which actions yield the highest rewards.

2. Overview of Q-learning:

- Q-learning is a model-free RL algorithm that allows an agent to learn the value of actions in given states without needing a model of the environment.
- The core idea is to learn a function Q(s, a) that estimates the expected future rewards for taking action ‘a’ in state ‘s’.
- Q-learning uses the Bellman equation to update the Q-values based on the reward received after taking an action and the maximum future rewards obtainable from the next state.

3. The Tic-Tac-Toe Game:

- Tic-Tac-Toe is a simple yet illustrative game for understanding RL concepts. It is played on a 3x3 grid where two players alternate placing their markers (X or O).
- The objective is to align three of one's markers horizontally, vertically, or diagonally.
- The game has a finite state space, making it suitable for Q-learning, allowing the algorithm to explore strategies effectively.

Theory

1. Q-learning Algorithm:

1. Q-learning Algorithm:

- The Q-learning algorithm is typically implemented using the following steps:
    1. **Initialize** the Q-values Q(s,a) arbitrarily for all state-action pairs.
    2. For each episode (game):
        - Initialize the starting state ‘s’.
        - Repeat until the game ends:
            1. Choose an action ‘a’ using a policy derived from Q (often using ε-greedy strategy).
            2. Take the action, observe the reward ‘r’ and the new state ‘s'’.
            3. Update the Q-value using the update rule:

Q(s,a)←Q(s,a)+α[r+γ a ′ max Q(s ′ ,a ′)−Q(s,a)]

where:

- α is the learning rate.
- γ is the discount factor.

4. Set the new state s′=s and repeat.

2. Key Components:

- **State Representation:** Each state of the Tic-Tac-Toe board can be represented as a unique configuration of the 9 cells (e.g., empty, X, O).
- **Action Space:** The possible actions correspond to placing an X or O in an empty cell.
- **Reward Structure:** The agent receives a positive reward for winning, a negative reward for losing, and zero for a draw or continuing the game without a win/loss.

3. ε-Greedy Policy:

- The ε-greedy policy balances exploration and exploitation by allowing the agent to explore random actions with probability ε, while choosing the best-known action with probability 1−ε1 - ε1−ε.
- This helps in discovering new strategies while still leveraging known good actions.

4. Learning Rate and Discount Factor:

- The learning rate (α) determines how much new information overrides old information. A higher rate means faster learning, but it may lead to instability.
- The discount factor (γ) determines the importance of future rewards. A value close to 1 makes the agent consider long-term rewards, while a value closer to 0 makes it focus on immediate rewards.

Methodology

1. Environment Setup:

- **Game Representation:**
    - Represent the Tic-Tac-Toe board as a 3x3 grid using a 2D array or list. Each cell can be assigned a value:
        - 0 for an empty cell,

      - 1 for an X,
      - -1 for an O.
    - The state of the game can be represented as a single list or array that holds these values.

2. State and Action Space Definition:

- **State Space:**
    - Define all possible board configurations. Since there are 9 cells and each can be empty, X, or O, the theoretical state space is 393^939 (though many will not be valid game states).
- **Action Space:**
    - The actions correspond to placing a marker (X or O) in an empty cell. The available actions can be dynamically identified based on the current state of the board.

3. Q-value Initialization:

- Initialize a Q-table (a dictionary or 2D array) where each state-action pair is stored with an initial Q-value, usually set to zero.

4. Hyperparameter Configuration:

- Define hyperparameters for the Q-learning algorithm:
    - **Learning Rate (α)**: A typical starting point is 0.1.
    - **Discount Factor (γ)**: A value around 0.9 is commonly used.
    - **Exploration Rate (ε)**: Start with a value (e.g., 1.0) and decay it over time to reduce exploration as the agent learns.

5. Training Process:

- The training process consists of running multiple episodes of the game to allow the agent to learn from its experiences.

6. Reward Structure:

- Define the reward mechanism based on the game's outcome:
    - +1 for winning,
    - -1 for losing,
    - 0 for a draw or if the game continues.
- You can also provide small penalties for each move to encourage shorter games.

7. Decay Exploration Rate:

- Implement an exploration decay mechanism that reduces ε over time, encouraging the agent to exploit learned strategies as it gains more experience.

8. Testing and Evaluation:

- After training, evaluate the performance of the trained Q-learning agent:
    - Play a series of games against a fixed opponent (which could be a random or a simple strategy-based player).
    - Measure performance using win/loss ratios, average number of moves until a win, and stability of learned strategies.
- Optionally, analyze the Q-values to understand the learned strategies for different board states.

Experimenting and Training

1. Experiment Design:

- **Objective:** The primary goal is to train a Q-learning agent to play Tic-Tac-Toe and assess its performance over time.
- **Control Variables:**
    - Fixed learning rate (α),
    - Discount factor (γ),
    - Initial exploration rate (ε),
    - Number of episodes for training.
- **Independent Variables:**
    - The decay rate of the exploration factor,
    - Different opponents (random player, strategic player),
    - The number of training episodes.

2. Training Setup:

- **Environment Configuration:**
    - Implement the Tic-Tac-Toe game logic, including state transitions and the evaluation of win/loss conditions.
- **Q-learning Agent Implementation:**
    - Use the previously discussed Q-learning algorithm.

3. Running Experiments:

- **Baseline Training:**
    - Start with a baseline configuration (fixed α, γ, and ε) and run the Q-learning algorithm for a predefined number of episodes (e.g., 10,000).
    - Track the agent's performance metrics, such as win/loss ratio, average number of moves per game, and changes in the Q-values over time.
- **Variable Exploration Rates:**
    - Experiment with different ε decay rates (e.g., linear vs. exponential decay) to observe how it impacts the agent's learning and performance.
- **Opponent Variation:**
    - Test the trained agent against different types of opponents:
        - A random player (for initial assessments of learning).
        - A simple heuristic player that follows a basic strategy (e.g., prioritizing corners and center).

Conclusion

In this report, we explored the development and application of a Q-learning agent for playing Tic-Tac-Toe, demonstrating how reinforcement learning can enable an agent to learn optimal strategies through repeated interactions with the game environment. By breaking down the problem, we designed a systematic approach to represent the game's state and action spaces, configured the Q-learning algorithm, and trained the agent through a series of experiments.

The Q-learning algorithm proved effective in learning successful strategies for Tic-Tac-Toe, as evidenced by the improvement in the agent's performance over successive training episodes. Through experimentation, we observed how adjusting hyperparameters, such as the learning rate, discount factor, and exploration rate, affected the agent's ability to balance exploration and exploitation, ultimately influencing the quality of the learned strategies.

Our experiments demonstrated that with sufficient training, the agent could consistently perform well against random opponents and even against heuristic-based players. The results underscored the importance of reward structure and exploration decay, as these factors directly impacted the agent's adaptability and ability to generalize across different game situations.

Key Takeaways

1. **Effectiveness of Q-learning:**

   - Q-learning successfully enables an agent to learn optimal strategies in environments with well-defined states and rewards, like Tic-Tac-Toe.

2. **Importance of Hyperparameters:**
   - The learning rate, discount factor, and exploration strategy significantly influence the agent's learning progress and performance, especially in balancing short-term vs. long-term rewards.
3. **Applicability to Larger Problems:**
   - While Tic-Tac-Toe serves as a simple introductory example, this framework can be adapted to more complex environments by enhancing state representations and extending Q-learning with techniques like deep Q-learning.

References

- Python pickle library documentation:

  https://docs.python.org/3/library/pickle.html#pickle.dump
- Research done regarding Q-learning:

  https://www.geeksforgeeks.org/q-learning-in-python/

Appendix

Flow-chart

```
Start
 |
 |-- Load Q-table
 |     |
 |     |-- [Q-table exists] --> Load Q-table from file
 |     |-- [No Q-table] --> Initialize an empty Q-table
 |
 |-- Training Loop (for each episode)
```

```
|     |
|     |-- Initialize board (3x3 empty grid)
|     |
|     |-- Game Loop (until game is over)
|          |
|          |-- Agent Action
|          |      |
|          |      |-- Choose action using epsilon-greedy policy
|          |      |-- Update board with chosen move
|          |      |-- Check for Win/Tie
|          |      |     |
|          |      |     |-- [Agent wins] --> Update Q-table with reward = +1 --> End Game
|          |      |     |-- [Tie] --> Update Q-table with reward = +0.5 --> End Game
|          |      |     |-- [No win/tie] --> Update Q-table with reward = 0 --> Opponent Move
|          |
|          |-- Opponent Action (random move)
|               |
|               |-- Update board with opponent's move
|               |-- Check for Win/Tie
|               |     |
|               |     |-- [Opponent wins] --> Update Q-table with reward = -1 --> End Game
|               |     |-- [Tie] --> Update Q-table with reward = +0.5 --> End Game
|
|-- Save Q-table to file after training
|
|-- Play Game (Agent vs Random Opponent)
|     |
|     |-- Game Loop (until game is over)
|          |
 |          |-- Agent Move (choose best action based on Q-table)
 |          |      |
 |          |      |-- Check for Win/Tie
 |          |           |
 |          |           |-- [Agent wins] --> Print "Agent wins!" --> End Game
 |          |
 |          |-- Opponent Move (random move)
 |               |
 |               |-- Check for Win/Tie
 |                    |
 |                    |-- [Opponent wins] --> Print "Opponent wins!" --> End Game
 |                    |-- [Tie] --> Print "It's a tie!" --> End Game
 |
End
```

Complete code

```
import numpy as np
import random
from collections import defaultdict
import pickle

# Initialize game variables and state
def initialize_board():
   return np.full((3, 3), ' ')

def get_state(board):
   return tuple(board.flatten())

def available_moves(board):
   return [(i, j) for i in range(3) for j in range(3) if board[i, j] == ' ']

def make_move(board, row, col, player):
   board[row, col] = player
   return board

def check_winner(board, row, col, player):
   # Check row, column, and diagonals
   if all([board[row, c] == player for c in range(3)]) or \
      all([board[r, col] == player for r in range(3)]) or \
      (row == col and all([board[i, i] == player for i in range(3)])) or \
      (row + col == 2 and all([board[i, 2 - i] == player for i in range(3)])):
       return True
   return False

def is_full(board):
   return ' ' not in board

# Q-learning functions
def initialize_q_table():
   return defaultdict(float)

def choose_action(state, q_table, available_moves, epsilon=0.1):
   if random.random() < epsilon:
       return random.choice(available_moves)
   q_values = [q_table[(state, move)] for move in available_moves]
   max_q = max(q_values)
   return random.choice([move for move, q in zip(available_moves, q_values) if q == max_q])
```

```
def update_q_table(q_table, state, action, reward, next_state, next_available_moves, alpha=0.1,
gamma=0.9):
    old_q = q_table[(state, action)]
    next_max_q = max([q_table[(next_state, move)] for move in next_available_moves], default=0)
    q_table[(state, action)] = old_q + alpha * (reward + gamma * next_max_q - old_q)



# Add Q-table save and load functions
def save_q_table(q_table, filename='q_table.pkl'):
    with open(filename, 'wb') as file:
        pickle.dump(q_table, file)
    print("Q-table saved to", filename)


def load_q_table(filename='q_table.pkl'):
    try:
        with open(filename, 'rb') as file:
            q_table = pickle.load(file)
        print("Q-table loaded from", filename)
        return q_table
    except FileNotFoundError:
        print("Q-table file not found, starting with an empty Q-table.")
        return defaultdict(float)


# Training function
def train(q_table, episodes=5000):
    print('Training Phase: ')
    for _ in range(episodes):
        board = initialize_board()
        state = get_state(board)
        done = False
        while not done:
            moves = available_moves(board)
            action = choose_action(state, q_table, moves)
            board = make_move(board, action[0], action[1], 'X')
            next_state = get_state(board)

            if check_winner(board, action[0], action[1], 'X'):
                update_q_table(q_table, state, action, 1, next_state, [])
                done = True
            elif is_full(board):
                update_q_table(q_table, state, action, 0.5, next_state, [])
                done = True
            else:
                # Opponent (random) move
                opp_action = random.choice(available_moves(board))
                board = make_move(board, opp_action[0], opp_action[1], 'O')
                if check_winner(board, opp_action[0], opp_action[1], 'O'):
                    update_q_table(q_table, state, action, -1, next_state, [])
                    done = True
                elif is_full(board):
                    update_q_table(q_table, state, action, 0.5, next_state, [])
                    done = True
                else:
                    update_q_table(q_table, state, action, 0, next_state, available_moves(board))
            state = next_state

# Play function
def play(q_table):
    print('Play Phase: ')
    board = initialize_board()
    state = get_state(board)
    while True:
        moves = available_moves(board)
        action = choose_action(state, q_table, moves, epsilon=0)  # Choose best action
        board = make_move(board, action[0], action[1], 'X')
        print_board(board)
        if check_winner(board, action[0], action[1], 'X'):
            print("Agent wins!")
            break
        elif is_full(board):
            print("It's a tie!")
            break

        row, col = map(int, input("Enter your move (row col, 0-2): ").split())
        board = make_move(board, row, col, 'O')
        print_board(board)
        if check_winner(board, row, col, 'O'):
            print("You win!")
            break
        elif is_full(board):
            print("It's a tie!")
            break
        state = get_state(board)

# Print the board
def print_board(board):
    for row in board:
        print(' | '.join(row))
        print('---------')

q_table = load_q_table()  # Load existing Q-table if available
train(q_table)            # Train the agent
save_q_table(q_table)     # Save the trained Q-table
play(q_table)             # Play against the agent
```