Homework 3: Tic Tac Toe

Hai Dinh (19960331-4494)

1. Presentation of results

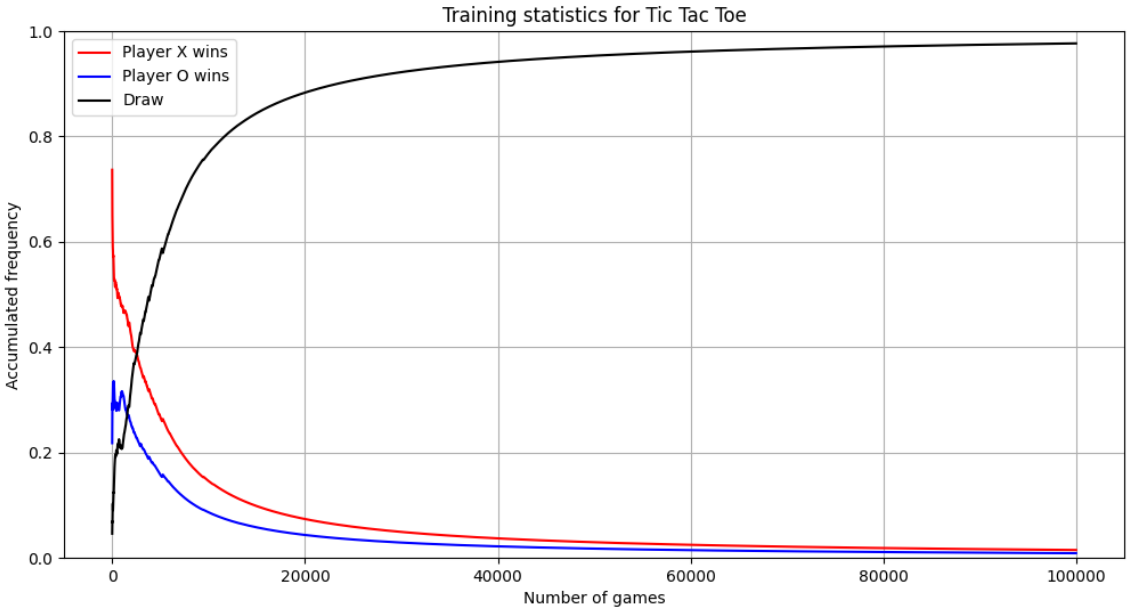

Figure 1: Learning curves smoothed over a window of 30. The training used $\alpha = 0.1$, $\gamma = 1$, $\varepsilon = 0.3$ (which is kept the same for the first 10^3 episodes, and then is decreased by a factor 0.9 after every 100th episode). The train runs for 10^5 episodes.

Implementation:

- The `State` class represents a single entry in the Q-table. It's basically a combination of a board configuration and its corresponding Q-values. The class is also responsible for update the Q-values based on the SARSA equation, make epsilon-greedy move for a specific player, and check if the game has ended and who has won.
- The `Player` class represents a single player. Each player stores all the states that it has seen so far across all the games. The player also stores all the states that it has seen so far in the current game together with the chosen moves, such that the player can then update all the Q-values of these states at the end of each game.
- For training, we loop through all the games, and for each game, each player takes its turn until the game has reached its terminal state. Only then will the players update Q-values of their states.

Analysis of the results:

- In the beginning, there are not so many draws. But as the training goes on, both players get better and better, and thus by the end of the training, almost every game ends in a draw.
- Since we are using epsilon decay techniques (in which ε is decayed towards 0 as the training goes on), in the beginning, the players are allowed to explore more, instead of always making the moves with the highest future rewards known so far. During this exploration phase, moves are made more randomly, and thus it's more random on who's gonna win. However, as training goes on, the moves made by the players are more and more greedy (i.e., both players are trying to play at the best ability), thus draws happen more often.

2. Code

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
from progressbar import progressbar # pip install progressbar2

# Global constants
alpha = 0.1
gamma = 1
eps_initial = 0.5
eps_decay = 0.9
eps_decay_freq = 100
eps_decay_start = int(1e3)
n_games = int(1e5)
window_size = 30


class State:
    """
    State is a combination of a board configuration and its corresponding Q values.
    """

    def __init__(self, board=np.zeros(shape=(9,), dtype=np.int8)):
        """
        Initialize a state using a numpy board configuration.
        """
        self.board = board
        self.id = str(board)
        self.q_values = np.zeros(board.shape, dtype=np.float64)
        self.q_values[board != 0] = np.nan

    def to_numpy(self):
        """
        Return a 6-by-3 numpy array containing both the board configuration and its Q values.
        """
        board = self.board.reshape(3, 3)
        q_values = self.q_values.reshape(3, 3)
        return np.concatenate([board, q_values], axis=0)

    def is_filled(self):
        """
        Check if the entire board has been filled.
        """
        return 0 not in self.board

    def is_a_win(self, player):
        """
        Check if this state represents a win for a given player.
        """
        board = self.board.reshape(3, 3)
        winning_sum = 3 * player.id

        return (winning_sum in board.sum(axis=0)          # check columns
             or winning_sum in board.sum(axis=1)          # check rows
             or winning_sum == board.trace()              # check diagonal
             or winning_sum == np.fliplr(board).trace())  # check anti-diagonal

    def eps_greedy_move(self, player, eps):
```

```
        """
        Make an epsilon-greedy move for a given player. Return the chosen move, the new state to the
        opposite player, the reward for the move, and whether the game has reached a terminal state.
        """
        greedy_move = np.nanargmax(self.q_values)
        valid_moves = ~np.isnan(self.q_values)
        invalid_moves = np.isnan(self.q_values)

        eps_greedy_policy = np.ones(self.q_values.shape) * (eps / np.sum(valid_moves))
        eps_greedy_policy[greedy_move] += (1 - eps)
        eps_greedy_policy[invalid_moves] = 0

        chosen_move = np.random.choice(len(self.q_values), p=eps_greedy_policy)
        new_board = np.copy(self.board)
        new_board[chosen_move] = player.id
        new_state = State(board=new_board)

        reward = 1 if new_state.is_a_win(player) else 0
        terminal = reward == 1 or new_state.is_filled()
        return chosen_move, new_state, reward, terminal

    def update_q(self, move, reward, next_state):
        """
        Update the Q value of a specific move using the SARSA equation.
        """
        delta = alpha * (reward + gamma * np.nanmax(next_state.q_values) - self.q_values[move])
        self.q_values[move] += delta


class Player:
    """
    Representation of a player, storing all the states that the player has seen so far.
    """

    def __init__(self, player_name, player_id):
        """
        Create a new player with a specific ID.
        """
        self.name = player_name
        self.id = player_id
        self.q_table = {}       # stores all the states the player has seen so far across all games
        self.statistics = []    # stores win/loss/draw statistics for the player
        self.game_records = []  # records of all decisions made by a player during a game

    def get_average_accum_freq(self, result, window_size):
        """
        Get accumulated frequencies (average over a window) for a specific result (win/loss/draw).
        """
        stats = np.array(self.statistics)
        frequencies = np.cumsum(stats == result) / np.arange(1, len(stats)+1)
        return np.convolve(frequencies, np.ones(window_size) / window_size, mode="valid")

    def save_q_table(self, filepath):
        q_entries = [state.to_numpy() for state in self.q_table.values()]
        q_table = np.concatenate(q_entries, axis=1)
        np.savetxt(filepath, q_table, delimiter=",")
        print("Shape of Q table for {}: {}".format(self.name, q_table.shape))

    def find_stored_state(self, state):
```

```
        """
        Find a stored state that has the same board configuration as the giaven state. If no such
        state is found, then save the given state and return the same state to the caller.
        """
        try:
            return self.q_table[state.id]
        except KeyError:
            self.q_table[state.id] = state
            return state

    def make_move(self, curr_state, chosen_move, reward):
        """
        Make a move by simply adding a new game record.
        """
        self.game_records.append([curr_state, chosen_move, reward])

    def end_game(self, final_reward):
        """
        Called when the game has ended. Final reward is +1 for a win, -1 for a loss, 0 for a draw.
        """
        self.game_records[-1][-1] = final_reward # update last record with the final reward
        self.statistics.append(final_reward) # add the game results to the statistics
        next_state = State() # initialize an empty state where all Q values are 0

        # Propagate backwards to update Q values of all states in this game
        for (curr_state, chosen_move, reward) in reversed(self.game_records):
            curr_state.update_q(chosen_move, reward, next_state)
            next_state = curr_state

        # Wipe away all records to prepare for the next game
        self.game_records = []


def train(players):
    """
    Train both players by letting them playing against each other in many games.
    """
    for game in progressbar(range(n_games)):
        turn = -1
        state = State()
        terminal = False

        if game < eps_decay_start:
            eps = eps_initial
        else:
            eps = eps_initial * (eps_decay ** int((game - eps_decay_start) / eps_decay_freq))

        while not terminal:
            turn = -turn # flip the turn
            player = players[turn]
            state = player.find_stored_state(state)
            chosen_move, new_state, reward, terminal = state.eps_greedy_move(player, eps)
            player.make_move(state, chosen_move, reward)
            state = new_state

        players[turn].end_game(reward)
        players[-turn].end_game(-reward)
```

```python
def main(args):
    """
    Main method responsible for training the players and save outputs to files.
    """
    players = {
        +1: Player("PlayerX", +1),
        -1: Player("PlayerO", -1),
    }
    train(players)
    players[+1].save_q_table(args.outdir + "/player1.csv")
    players[-1].save_q_table(args.outdir + "/player2.csv")

    plt.figure(figsize=(12, 6))
    plt.plot(players[+1].get_average_accum_freq(+1, window_size), "r-", label="Player X wins")
    plt.plot(players[+1].get_average_accum_freq(-1, window_size), "b-", label="Player O wins")
    plt.plot(players[+1].get_average_accum_freq(0, window_size), "k-", label="Draw")
    plt.title("Training statistics for Tic Tac Toe")
    plt.xlabel("Number of games")
    plt.ylabel("Accumulated frequency")
    plt.ylim(0, 1)
    plt.legend()
    plt.grid()
    plt.savefig(args.outdir + "/learning_curve.png")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simulate Tic Tac Toe with Q-Learning")
    parser.add_argument("--outdir", "-o", type=str, default=".", help="Out directory")
    args = main(parser.parse_args())
```