Tic Tac Toe

Eric Blohm

October 31, 2024

Here is the Python code for Tic Tac Toe task:

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


#check if game over, and return which player won
def CheckGameOver(board):
    ## check if any positions are open.
    if(0 not in board):
        return True, 0

    ## check horisontal
    for row in range(0,len(board)):
        sum = np.sum(board[row])
        if (sum == 3):
            return True, 1
        if(sum == -3):
            return True, -1

    ## check vertical
    for col in range(0,len(board[0])):
        sum = np.sum(board[:,col])
        if (sum == 3):
            return True, 1
        if(sum == -3):
            return True, -1

    ## check diagonal ##
    #p1
    if(board[0][0] == board[1][1] == board[2][2] == 1):
        return True, 1
    #p2
    if(board[0][0] == board[1][1] == board[2][2] == -1):
        return True,-1

    #p1
    if(board[0][2] == board[1][1] == board[2][0]== 1):
        return True, 1
    #p2
    if(board[0][2] == board[1][1] == board[2][0]== -1):
        return True, -1
    ######################

    return False, 0

def MoveAgent(board,Q,epsilon,p):
    new_board = board.copy()
    b = new_board.tobytes()
    pos = []
    if b in Q:
        r = np.random.rand()
        if(r < (1-epsilon)):
            pos = BestMove(new_board,Q)
            new_board[pos[0]][pos[1]] = p
            return new_board, pos
    #init Q-table if we havent seen this state before
    elif not(b in Q):
        tmp1 = np.zeros((3,3))
        tmp = setNan(new_board,tmp1)
        Q[b] = tmp

    zero_indices = np.argwhere(new_board == 0)
    pos = zero_indices[np.random.randint(zero_indices.shape[0])]

    new_board[pos[0]][pos[1]] = p
    return new_board, pos

#Return highest Q-value for the state "board".
def BestMove(board,Q):
    if(0 in board):
        b = board.tobytes()
        if (b in Q):
            action = np.nanmax(Q[b])
            pos = np.argwhere(action == Q[b])
            if (isinstance(pos, np.ndarray)):
                pos = pos[np.random.randint(pos.shape[0])]
                return pos
            return pos
    return None

#Prev state refers to state at t-2 or t-1, board is current state
def UpdateQ(board, prev_state, prev_action, Q, alpha, r_p, gameover):
    b_prev = prev_state.tobytes()
    prev_tmp = Q[b_prev]
    if(not gameover):
        b = board.tobytes()
        current_tmp = Q[b]
        best_pos = BestMove(board,Q)
        prev_tmp[prev_action[0]][prev_action[1]] += alpha* ( r_p + current_tmp[best_pos[0]][best_pos[1]] - prev_tmp[prev_action[0]][prev_action[1]])
    else:
        prev_tmp[prev_action[0]][prev_action[1]] += alpha* ( r_p - prev_tmp[prev_action[0]][prev_action[1]])
    Q[b_prev] = prev_tmp


def setNan(board,q):
    for i in range(0,len(board)):
        for j in range(0,len(board[0])):
            if(board[i][j] !=0):
                q[i][j] = np.nan
    return q


## convert from byte states back to board representation
def from_byte_to_board(bytes):
    shape = (3,3)
    dtype = 'float64'
    board_arr = np.frombuffer(bytes, dtype=dtype)
    return board_arr.reshape(shape)


## save as 2 x n matrix
def convert_states(Q):
    boards = []
    q_values = []
    for state in Q:
        board = from_byte_to_board(state)
        q_value = Q[state]
        boards.append(board)
        q_values.append(q_value)
    return [boards,q_values]


## Concatenate all boards and q values horizontally, so we get 6 rows and n columns.
def save_to_csv(player,name):
    expanded_rows = []
    for row in player:
        concatenated_row = np.hstack(row)
        expanded_rows.append(concatenated_row)

    df = pd.DataFrame(np.vstack(expanded_rows))
    df.to_csv(name, index=False, header=False, na_rep='NaN')

    print("Saved to ",name)


def main():

    #### Set parameters ####
    p1 = 1 # 'X'
    p2 = -1 # 'O'

    Q_p1 = {}
    Q_p2 = {}
    epsilon = 1
    decay_rate = 0.95
    alpha = 0.1
    K = 100000

    freq_p1 = 0
    freq_p2 = 0
    freq_draw = 0

    draw_probabilities = []
    win_p1_probabilities = []
    win_p2_probabilities = []
    rounds = []

    # Calculate draw probability every "round_interval" rounds
    round_interval = 250

    for k in range(0, K):
        if k > 20000 and k % 100 == 0:
            epsilon *= decay_rate

        board = np.zeros((3,3))
        board_states = [board]
        actions = []
        ## PLayer 1 always start ##
        current_p = p1
        gameOver = False
        t=0
        winner = 0
        ## Current game ##
        while(not gameOver):
            #board is the next state, action is paired with the current state.
            if current_p == p1:
                board, action = MoveAgent(board_states[t],Q_p1,epsilon,p1)
            elif current_p == p2:
                board, action = MoveAgent(board_states[t],Q_p2,epsilon,p2)

            actions.append(action)
            gameOver,winner = CheckGameOver(board)

            if t > 1 and (not gameOver):
                if(current_p == p1):
                    UpdateQ(board_states[t], board_states[t-2], actions[t-2], Q_p1, alpha, 0, gameOver)
                elif(current_p == p2):
                    UpdateQ(board_states[t], board_states[t-2], actions[t-2], Q_p2, alpha, 0, gameOver)

            ### board_states has one more state than actions. The +1 state will be the ending one.
            board_states.append(board)
            t+=1
            #Dont change player the final round.
            if not gameOver:
                current_p *= -1
        #################
        ### Update rewards ###
        if(winner == p1):
            #reward with 1
            UpdateQ(None, board_states[t-1], actions[t-1], Q_p1, alpha, 1, gameOver)
            #penalize with -1
            UpdateQ(None, board_states[t-2], actions[t-2], Q_p2, alpha, -1, gameOver)
            freq_p1 +=1

        elif(winner == p2):
            #reward with 1
            UpdateQ(None, board_states[t-1], actions[t-1], Q_p2, alpha, 1, gameOver)
            #penalize with -1
            UpdateQ(None, board_states[t-2], actions[t-2], Q_p1, alpha, -1, gameOver)
            freq_p2 +=1
        elif winner==0:
            if current_p == p1:
                UpdateQ(None, board_states[t-1], actions[t-1], Q_p1, alpha, 0, gameOver)
                UpdateQ(None, board_states[t-2], actions[t-2], Q_p2, alpha, 0, gameOver)
            elif current_p == p2:
                UpdateQ(None, board_states[t-1], actions[t-1], Q_p2, alpha, 0, gameOver)
                UpdateQ(None, board_states[t-2], actions[t-2], Q_p1, alpha, 0, gameOver)
            freq_draw +=1
        #####################

        ## Calculate probabilities, for plotting.
        ## Do an average over "round interval" games and save these points. Then reset the frequencies.
        if k != 0 and k % round_interval == 0:
            draw_prob = freq_draw / round_interval
            win_prob_p1 = freq_p1 / round_interval
            win_prob_p2 = freq_p2 / round_interval
            draw_probabilities.append(draw_prob)
            win_p1_probabilities.append(win_prob_p1)
            win_p2_probabilities.append(win_prob_p2)
            ## reset frequencies
            freq_p1 = 0
            freq_p2 = 0
            freq_draw = 0
            rounds.append(k)

    plt.figure(figsize=(10, 6))
    plt.plot(np.array(rounds)/1000, draw_probabilities, label="Draw probability")
    plt.plot(np.array(rounds)/1000, win_p1_probabilities, label="P1 win probability")
    plt.plot(np.array(rounds)/1000, win_p2_probabilities, label="P2 win probability")

    plt.xlabel("Number of rounds x $10^3$")
    plt.ylabel("Probability")
    plt.ylim([-0.1,1.1])
    plt.title("Probabilities for wins and draw")
    plt.legend()
    plt.grid(True)
    plt.show()

    print(f"Length of dictionaries: Q1 {len(Q_p1)}, Q2 {len(Q_p2)}")

    print("K: ", K)

    player1 = convert_states(Q_p1)
    player2 = convert_states(Q_p2)


    save_to_csv(player1,’player1.csv’)
    save_to_csv(player2,’player2.csv’)




if __name__ == "__main__":
    main()
```