Tic Tac Toe
FFR135 - HW3.4
Navid Mousavi

For this problem, I implemented Q-learning algorithm as described on OpenTA with parameters $\alpha = 0.1$, $\gamma = 1$, and used a decaying ϵ starting with $\epsilon_0 = 1$ and decreasing with a factor of 0.95 after each 100 games. I set $\epsilon = 0$ after reaching 0.01. Different Q-table is used for each player. I stopped training after 3×10^4 episodes and as it can be seen in Figure 1 the winning rate of both players converges to zero and all games end in draw after around 16000 games were played, which is evidence of having two perfect players, since none of them looses.

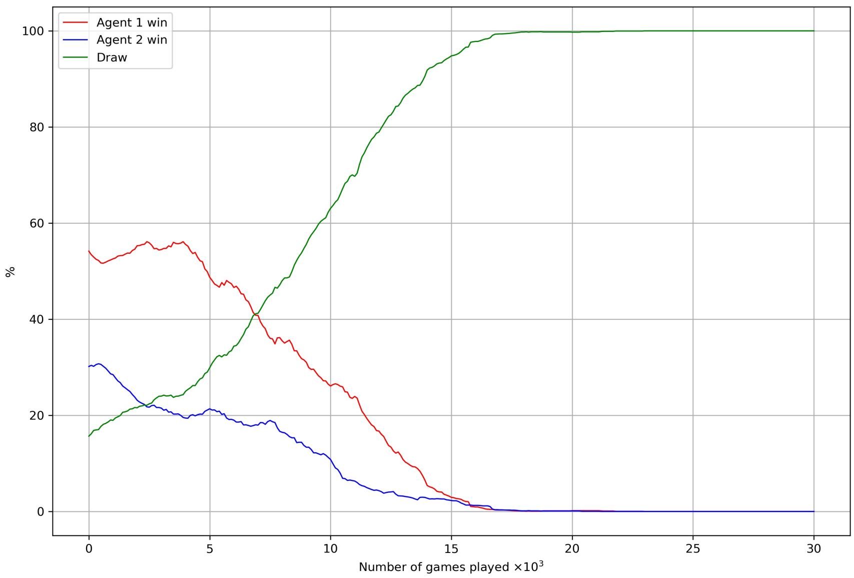

Fig. 1: Result of each 100 games, averaged over a moving window with size 3000 games. In the beginning agents play completely randomly ($\epsilon = 1$) and as expected starting player has a higher probability to win the games. After a few thousand games both players improve and the highest reward they can get is ending the game in draw.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 27 07:15:35 2020

@author: navid
"""
import numpy as np
import h5py
import time

def initialize_borad():
    board = np.zeros((3,3))
    return board

def get_state(Q , board):
    for i in range(len(Q)):
        if np.array_equal(Q[i][0], board):
            return i , Q
        else:
            if i == len(Q) - 1:
                allowed_moves = remove_filled(board)
                Q = np.append(Q,[[board , allowed_moves]],axis=0)
                for rotation in range(1,4):
                    if np.array_equal(np.rot90(board,rotation) , board):
                        pass
                    else:
                        Q = np.append(Q,[[np.rot90(board,rotation) , np.rot90(
                            allowed_moves,rotation)]],axis=0)

                return i+1 , Q

def get_move(Q , state , epsilon):
    if np.random.uniform() > epsilon:
        possible_moves = np.where(Q[state][0] == 0)
        possible_moves = list(zip(possible_moves[0],possible_moves[1]))
        move = possible_moves[0]
        for i in range(1,len(possible_moves)):
            if Q[state][1][move] < Q[state][1][possible_moves[i]]:
                move = possible_moves[i]
        return move
    else:
        possible_moves = np.where(Q[state][0] == 0)
        possible_moves = list(zip(possible_moves[0],possible_moves[1]))
        move = possible_moves[np.random.choice(len(possible_moves))]
        return move


def remove_filled(board):
    a = np.ones((3,3))
    filled = np.where(board != 0)
    filled = list(zip(filled[0],filled[1]))
    for i in range(len(filled)):
        a[filled[i]] = np.nan
    return a

def eval_game(board , player):
    if 0 in board:
        for i in range(3):
```

```
            if board[i,0] == player and board[i,1] == player and board[i,2] ==
                player:
                end_game = True
                winner = player
                break
            elif board[0,i] == player and board[1,i] == player and board[2,i] ==
                 player:
                end_game = True
                winner = player
                break
            elif board[0,0] == player and board[1,1] == player and board[2,2] ==
                 player:
                end_game = True
                winner = player
            elif board[2,0] == player and board[1,1] == player and board[0,2] ==
                 player:
                end_game = True
                winner = player
            else:
                end_game = False
                winner = 0
    else:
        end_game = True
        for i in range(3):
            if board[i,0] == player and board[i,1] == player and board[i,2] ==
                player:
                winner = player
                break
            elif board[0,i] == player and board[1,i] == player and board[2,i] ==
                 player:
                winner = player
                break
            elif board[0,0] == player and board[1,1] == player and board[2,2] ==
                 player:
                winner = player
            elif board[2,0] == player and board[1,1] == player and board[0,2] ==
                 player:
                winner = player
            else:
                winner = 0

    return winner , end_game

def update_Q(Q , state , new_state , action , R , end_game):
    if end_game:
        max_estimate = 0
    else:
        possible_moves = np.where(Q[new_state][0] == 0)
        possible = list(zip(possible_moves[0],possible_moves[1]))
        max_estimate = Q[new_state][1][possible[0]]
        for i in range(1,len(possible)):
            if max_estimate < Q[new_state][1][possible[i]]:
                max_estimate = Q[new_state][1][possible[i]]
    Q[state][1][action] = Q[state][1][action] + alpha*(R + (gamma*max_estimate)
      - Q[state][1][action])
    dummy = np.copy(Q[state][0])
    for rotation in range(1,4):
        rotated_board = np.rot90(dummy, rotation)
        rotated_state , Q = get_state(Q, rotated_board)
        Q[rotated_state][1] = np.rot90(Q[state][1],rotation)
```

```python
    return Q

def get_move_random(board):
    possible_moves = np.where(board == 0)
    possible_moves = list(zip(possible_moves[0],possible_moves[1]))
    move = possible_moves[np.random.choice(len(possible_moves))]
    return move


def print_board(board):
    for i in range(3):
        printable = []
        for j in range(3):
            if board[i,j] == 0:
                printable.append('-')
            elif board[i,j] == 1:
                printable.append('x')
            else:
                printable.append('o')
        print(f'{printable[0]} \t {printable[1]} \t {printable[2]}')



data = h5py.File('data-two-AI.h5','w')

Q1 = np.ones((1,2,3,3))
Q2 = np.ones((1,2,3,3))
epsilon = 1
alpha = 0.1
gamma = 1

game_number = 0
player1_wins = 0
player2_wins = 0
draw = 0

win_rate1 = []
win_rate2 = []
draw_rate = []
epsilon_list = []
player1_prev_state = 0
player2_state = 0
player2_move = 0
player1_wins = 0
player2_wins = 0
draw = 0
while game_number < 30000:
    if game_number%100 == 0:
        #print(f'after {game_number} games: player 1 wins = {player1_wins/10}% - player 2 wins = {player2_wins/10}% - draw = {draw/10}% - lenQ1 = {len(Q1)} - lenQ2 = {len(Q2)}')
        if epsilon > 0.01:
            epsilon = 0.95*epsilon
        else:
            epsilon = 0
        with open('log-two-AI.txt','a+') as log:
            log.write(f'after {game_number} games: player 1 wins = {player1_wins}% - player 2 wins = {player2_wins}% - draw = {draw}% - lenQ1 = {len(Q1)} - lenQ2 = {len(Q2)}\n')
```

```
    board = initialize_borad()
    end_game = False

    player1_state , Q1 = get_state(Q1, board)
    player1_move = get_move(Q1, player1_state, epsilon)
    board[player1_move] = 1
    player2_state , Q2 = get_state(Q2, board)
    player2_move = get_move(Q2 , player2_state , epsilon)
    board[player2_move] = -1
    player1_new_state , Q1 = get_state(Q1, board)
    Q1 = update_Q(Q1, player1_state, player1_new_state, player1_move, 0 , False)
    player1_state = player1_new_state
    while not end_game:
        player1_move = get_move(Q1, player1_state, epsilon)
        board[player1_move] = 1
        player2_new_state , Q2 = get_state(Q2, board)
        winner , end_game = eval_game(board,1)
        if end_game:
            break
        else:
            Q2 = update_Q(Q2, player2_state, player2_new_state, player2_move, 0
                , False)
            player2_state = player2_new_state
        player2_move = get_move(Q2 , player2_state , epsilon)
        board[player2_move] = -1
        player1_new_state , Q1 = get_state(Q1, board)
        winner , end_game = eval_game(board,-1)
        if end_game:
            break
        else:
            Q1 = update_Q(Q1, player1_state, player1_new_state, player1_move, 0
                , False)
            player1_state = player1_new_state


    game_number += 1
    if winner == 1:
        player1_wins += 1
        Q1 = update_Q(Q1, player1_state, player1_new_state, player1_move, 1 ,
            True)
        Q2 = update_Q(Q2, player2_state, player2_new_state, player2_move, -1 ,
            True)
    elif winner == -1:
        player2_wins += 1
        Q1 = update_Q(Q1, player1_state, player1_new_state, player1_move, -1 ,
            True)
        Q2 = update_Q(Q2, player2_state, player2_new_state, player2_move, 1 ,
            True)
    else:
        draw += 1
        Q1 = update_Q(Q1, player1_state, player1_new_state, player1_move, 0 ,
            True)
        Q2 = update_Q(Q2, player2_state, player2_new_state, player2_move, 0 ,
            True)
    epsilon_list.append(epsilon)
    win_rate1.append(player1_wins)
    win_rate2.append(player2_wins)
    draw_rate.append(draw)

data.create_dataset('Q1', data = Q1)
```

```
data.create_dataset('Q2', data = Q2)
data.create_dataset('win_rate1', data = np.array(win_rate1))
data.create_dataset('win_rate2', data = np.array(win_rate2))
data.create_dataset('draw_rate', data = np.array(draw_rate))
data.create_dataset('epsilon', data = np.array(epsilon_list))
print(f'total games played: {game_number}')
print(f'player 1 wins: {player1_wins}')
print(f'player 2 wins: {player2_wins}')
print(f'Draws: {draw}')

data.close()
```