

Tic Tac Toe

Eric Blohm

October 31, 2024

Here is the Python code for Tic Tac Toe task:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5
6 #check if game over, and return which player won
7 def CheckGameOver(board):
8     ## check if any positions are open.
9     if(0 not in board):
10         return True, 0
11
12     ## check horisontal
13     for row in range(0,len(board)):
14         sum = np.sum(board[row])
15         if (sum == 3):
16             return True, 1
17         if(sum == -3):
18             return True, -1
19
20     ## check vertical
21     for col in range(0,len(board[0])):
22         sum = np.sum(board[:,col])
23         if (sum == 3):
24             return True, 1
25         if(sum == -3):
26             return True, -1
27
28     ## check diagonal ##
29     #p1
30     if(board[0][0] == board[1][1] == board[2][2] == 1)
31         :
32         return True, 1
33     #p2
```

```

33     if(board[0][0] == board[1][1] == board[2][2] ==
34         -1):
35         return True, -1
36
37     #p1
38     if(board[0][2] == board[1][1] == board[2][0] == 1):
39         return True, 1
40
41     #p2
42     if(board[0][2] == board[1][1] == board[2][0] == -1)
43         :
44         return True, -1
45     #####
46
47     return False, 0
48
49 def MoveAgent(board,Q,epsilon,p):
50     new_board = board.copy()
51     b = new_board.tobytes()
52     pos = []
53     if b in Q:
54         r = np.random.rand()
55         if(r < (1-epsilon)):
56             pos = BestMove(new_board,Q)
57             new_board[pos[0]][pos[1]] = p
58             return new_board, pos
59     #init Q-table if we havent seen this state before
60     elif not(b in Q):
61         tmp1 = np.zeros((3,3))
62         tmp = setNan(new_board,tmp1)
63         Q[b] = tmp
64
65     zero_indices = np.argwhere(new_board == 0)
66     pos = zero_indices[np.random.randint(zero_indices.
67         shape[0])]
68
69     new_board[pos[0]][pos[1]] = p
70     return new_board, pos
71
72 #Return highest Q-value for the state "board".
73 def BestMove(board,Q):
74     if(0 in board):
75         b = board.tobytes()
76         if (b in Q):
77             action = np.nanmax(Q[b])
78             pos = np.argwhere(action == Q[b])
79             if (isinstance(pos, np.ndarray)):

```

```

76         pos = pos[np.random.randint(pos.shape
77                                     [0])]
78         return pos
79     return pos
80     return None
81 #Prev state refers to state at t-2 or t-1, board is
82   current state
83 def UpdateQ(board, prev_state, prev_action, Q, alpha,
84   r_p, gameover):
85     b_prev = prev_state.tobytes()
86     prev_tmp = Q[b_prev]
87     if(not gameover):
88         b = board.tobytes()
89         current_tmp = Q[b]
90         best_pos = BestMove(board,Q)
91         prev_tmp[prev_action[0]][prev_action[1]] +=
92             alpha* ( r_p + current_tmp[best_pos[0]][
93                 best_pos[1]] - prev_tmp[prev_action[0]][
94                     prev_action[1]])
95     else:
96         prev_tmp[prev_action[0]][prev_action[1]] +=
97             alpha* ( r_p - prev_tmp[prev_action[0]][
98                 prev_action[1]])
99     Q[b_prev] = prev_tmp
100
101
102
103 def setNan(board,q):
104     for i in range(0,len(board)):
105         for j in range(0,len(board[0])):
106             if(board[i][j] !=0):
107                 q[i][j] = np.nan
108     return q
109
110
111 ## convert from byte states back to board
112   representation
113 def from_byte_to_board(bytes):
114     shape = (3,3)
115     dtype = 'float64'
116     board_arr = np.frombuffer(bytes, dtype=dtype)
117     return board_arr.reshape(shape)
118
119
120 ## save as 2 x n matrix
121 def convert_states(Q):

```

```

113     boards = []
114     q_values = []
115     for state in Q:
116         board = from_byte_to_board(state)
117         q_value = Q[state]
118         boards.append(board)
119         q_values.append(q_value)
120     return [boards, q_values]
121
122
123     ## Concatenate all boards and q values horizontally,
124     so we get 6 rows and n columns.
125 def save_to_csv(player, name):
126     expanded_rows = []
127     for row in player:
128         concatenated_row = np.hstack(row)
129         expanded_rows.append(concatenated_row)
130
131     df = pd.DataFrame(np.vstack(expanded_rows))
132     df.to_csv(name, index=False, header=False, na_rep=
133         'NaN')
134
135     print("Saved to ", name)
136
137
138 def main():
139     ##### Set parameters #####
140     p1 = 1 # 'X'
141     p2 = -1 # 'O'
142
143     Q_p1 = {}
144     Q_p2 = {}
145     epsilon = 1
146     decay_rate = 0.95
147     alpha = 0.1
148     K = 100000
149
150     freq_p1 = 0
151     freq_p2 = 0
152     freq_draw = 0
153
154     draw_probabilities = []
155     win_p1_probabilities = []
156     win_p2_probabilities = []
157     rounds = []

```

```

157
158     # Calculate draw probability every "round_interval
      " rounds
159     round_interval = 250
160
161     for k in range(0, K):
162         if k > 20000 and k % 100 == 0:
163             epsilon *= decay_rate
164
165             board = np.zeros((3,3))
166             board_states = [board]
167             actions = []
168             ## PLayer 1 always start ##
169             current_p = p1
170             gameOver = False
171             t=0
172             winner = 0
173             ## Current game ##
174             while(not gameOver):
175                 #board is the next state, action is paired
                  with the current state.
176                 if current_p == p1:
177                     board, action = MoveAgent(board_states
                        [t],Q_p1,epsilon,p1)
178                 elif current_p == p2:
179                     board, action = MoveAgent(board_states
                        [t],Q_p2,epsilon,p2)
180
181                 actions.append(action)
182                 gameOver,winner = CheckGameOver(board)
183
184                 if t > 1 and (not gameOver):
185                     if(current_p == p1):
186                         UpdateQ(board_states[t],
                            board_states[t-2], actions[t
                                -2], Q_p1, alpha, 0, gameOver)
187                     elif(current_p == p2):
188                         UpdateQ(board_states[t],
                            board_states[t-2], actions[t
                                -2], Q_p2, alpha, 0, gameOver)
189
190                 ### board_states has one more state than
                  actions. The +1 state will be the
                  ending one.
191                 board_states.append(board)
192                 t+=1

```

```

193         #Dont change player the final round.
194         if not gameOver:
195             current_p *= -1
196         #####
197         ### Update rewards ###
198         if(winner == p1):
199             #reward with 1
200             UpdateQ(None, board_states[t-1], actions[t-1], Q_p1, alpha, 1, gameOver)
201             #penalize with -1
202             UpdateQ(None, board_states[t-2], actions[t-2], Q_p2, alpha, -1, gameOver)
203             freq_p1 +=1
204
205         elif(winner == p2):
206             #reward with 1
207             UpdateQ(None, board_states[t-1], actions[t-1], Q_p2, alpha, 1, gameOver)
208             #penalize with -1
209             UpdateQ(None, board_states[t-2], actions[t-2], Q_p1, alpha, -1, gameOver)
210             freq_p2 +=1
211         elif winner==0:
212             if current_p == p1:
213                 UpdateQ(None, board_states[t-1], actions[t-1], Q_p1, alpha, 0, gameOver)
214                 UpdateQ(None, board_states[t-2], actions[t-2], Q_p2, alpha, 0, gameOver)
215             elif current_p == p2:
216                 UpdateQ(None, board_states[t-1], actions[t-1], Q_p2, alpha, 0, gameOver)
217                 UpdateQ(None, board_states[t-2], actions[t-2], Q_p1, alpha, 0, gameOver)
218             freq_draw +=1
219         #####
220
221         ## Calculate probabilities, for plotting.
222         ## Do an average over "round interval" games and save these points. Then reset the frequencies.
223         if k != 0 and k % round_interval == 0:
224             draw_prob = freq_draw / round_interval

```

```

225         win_prob_p1 = freq_p1 / round_interval
226         win_prob_p2 = freq_p2 / round_interval
227         draw_probabilities.append(draw_prob)
228         win_p1_probabilities.append(win_prob_p1)
229         win_p2_probabilities.append(win_prob_p2)
230         ## reset frequencies
231         freq_p1 = 0
232         freq_p2 = 0
233         freq_draw = 0
234         rounds.append(k)
235
236     plt.figure(figsize=(10, 6))
237     plt.plot(np.array(rounds)/1000, draw_probabilities
238             , label="Draw probability")
239     plt.plot(np.array(rounds)/1000,
240             win_p1_probabilities, label="P1 win probability")
241     plt.plot(np.array(rounds)/1000,
242             win_p2_probabilities, label="P2 win probability")
243
244     plt.xlabel("Number of rounds x  $10^{-3}$ ")
245     plt.ylabel("Probability")
246     plt.ylim([-0.1,1.1])
247     plt.title("Probabilities for wins and draw")
248     plt.legend()
249     plt.grid(True)
250     plt.show()
251
252     print(f"Length of dictionaries: Q1 {len(Q_p1)}, Q2 {len(Q_p2)}")
253
254     print("K: ", K)
255
256     player1 = convert_states(Q_p1)
257     player2 = convert_states(Q_p2)
258
259     save_to_csv(player1, 'player1.csv')
260     save_to_csv(player2, 'player2.csv')
261
262
263 if __name__ == "__main__":
264     main()

```

