

## TIC TAC TOE

Question ①

Let  $S$  be the set of states.

Enemy - X
Agent - O

∴ Each element of state  $S$  is a  $3 \times 3$  matrix.

(represented using list of numpy array)

$$S = \{S_t\}_{t=1}^n$$

$$S_t = \begin{bmatrix} s_{00} & s_{01} & s_{02} \\ s_{10} & s_{11} & s_{12} \\ s_{20} & s_{21} & s_{22} \end{bmatrix}$$

such that

$$s_{ij} = \begin{cases} -1 & \text{enemy move} \\ +1 & \text{agent move} \\ 0 & \text{empty} \end{cases}$$

In addition, for all non-terminal states

$$\sum_{i=0}^2 \sum_{j=0}^2 s_{ij} = -1$$

for terminal states

$$\sum_{i=0}^2 \sum_{j=0}^2 s_{ij} = 0 \quad \text{if agent wins}$$

$$\sum_{i=0}^2 \sum_{j=0}^2 s_{ij} = -1 \quad \text{if enemy wins or draw.}$$

python

→ Each state has been also represented in a string format

String representation of a random state = " --- -XXO- -"

'-' represents empty cell /  $s_{ij}=0$

↑  
python string

'X' represents enemy cell /  $s_{ij}=-1$

'O' represents agent or player cell /  $s_{ij}=1$

For example,

$$S_t = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix} = "X---OX---" = \begin{array}{|c|c|c|} \hline X & & \\ \hline & O & X \\ \hline & & \\ \hline \end{array}$$

Action set = {0, 1, 2, 3, 4, 5, 6, 7, 8}

Action = i

$0 \leq i \leq 8$  and  $i \in \mathbb{Z}$  (set of integers)

$\downarrow$

↳ corresponds to putting 'o' in the  $i^{\text{th}}$  location of the string state

for example, Action = 4  $\Rightarrow$  "----xox-" (Initial State)

$\downarrow$   
Action = 4

"----oxox-" (state after performing action = 4)

## 5 Terminal States where player/agent wins

$$S_t = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & -1 & 0 \\ \hline -1 & 0 & -1 \\ \hline \end{array} \equiv "000-x-x-x"$$

$$S_t = \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline -1 & 1 & 0 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \equiv "0-xx0-x-0"$$

$$S_t = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline -1 & 1 & -1 \\ \hline 0 & -1 & -1 \\ \hline \end{array} \equiv "000x0x-xx"$$

$$S_t = \begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & -1 & -1 \\ \hline 1 & 0 & 0 \\ \hline \end{array} \equiv "0-x0xx0--"$$

$$S_t = \begin{array}{|c|c|c|} \hline -1 & 1 & 0 \\ \hline -1 & 1 & -1 \\ \hline 1 & 1 & -1 \\ \hline \end{array} \equiv "xo-xox00x"$$

# CSE257\_A3\_Q1\_TicTacToe\_v1

December 5, 2021

## 0.0.1 TicTacToe Game

```
[286]: import math
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

```
[199]: class Simulator:
    def __init__(self):
        pass

    def has_empty_cells(self, board):
        result = True

        if np.any(board == 0) == False:
            result = False

        return result

    def find_empty_cells(self, board):
        empty_cells = []

        if self.has_empty_cells(board) == True:
            for i in range(3):
                for j in range(3):
                    if board[i,j] == 0:
                        empty_cells.append((i,j))

        return empty_cells

    def get_reward(self, board):
        reward = 0
        if self.has_agent_won(board) == True:
            reward = 10
        elif self.has_enemy_won(board) == True:
            reward = -10
        elif self.has_game_drawn(board) == True:
```

```

        reward = 0
    else:
        reward = 1

    return reward

def has_agent_won(self, board):
    result = False

    row_wise_sum = np.sum(board, axis=0)
    col_wise_sum = np.sum(board, axis=1)
    main_diag_sum = np.sum([board[i,i] for i in range(3)])
    off_diag_sum = np.sum([board[i,2-i] for i in range(3)])

    if np.any(row_wise_sum == 3) or np.any(col_wise_sum == 3) or
    ↪main_diag_sum == 3 or off_diag_sum == 3:
        result = True

    return result

def has_enemy_won(self, board):
    result = False

    row_wise_sum = np.sum(board, axis=0)
    col_wise_sum = np.sum(board, axis=1)
    main_diag_sum = np.sum([board[i,i] for i in range(3)])
    off_diag_sum = np.sum([board[i,2-i] for i in range(3)])

    if np.any(row_wise_sum == -3) or np.any(col_wise_sum == -3) or
    ↪main_diag_sum == -3 or off_diag_sum == -3:
        result = True

    return result

def has_game_drawn(self, board):
    result = False

    if self.has_empty_cells(board) == False and self.has_agent_won(board) or
    ↪== False and self.has_enemy_won(board) == False:
        result = True

    return result

def has_game_end(self, board):
    result = False

```

```

        if self.has_agent_won(board) == True or self.has_enemy_won(board) == True:
    ↪True or self.has_game_drawn(board) == True:
            result = True

        return result

# returns the updated board state and a bool specifying whether enemy moved
↪or not
def enemy_move(self, board, empty_cell):
    has_enemy_moved = False

#     if self.has_empty_cells(board) == False:
#         return board, has_enemy_moved

#     # find empty cells
#     empty_cells = self.find_empty_cells(board)
#     num_empty_cells = len(empty_cells)

#     # generate a random number between [0, num_empty_cells-1] (both
↪inclusive)
#     rand_num = np.random.randint(0, num_empty_cells)

#     enemy_move_pos = empty_cells[rand_num]

    board[empty_cell[0], empty_cell[1]] = -1
    has_enemy_moved = True

    return has_enemy_moved, board

# returns the updated board state and a bool specifying whether enemy moved
↪or not
def agent_move(self, board, empty_cell):
    has_agent_moved = False

    board[empty_cell[0], empty_cell[1]] = 1
    has_agent_moved = True

    return has_agent_moved, board

def compute_reward_to_go(self, traj, gamma=0.9):
    traj_len = len(traj)
    reward_to_go = []

    for i in range(traj_len):
        temp_sum = 0
        for j in range(i, traj_len):
            temp_sum = temp_sum + pow(gamma, j-i) * traj[j].reward

```

```

        reward_to_go.append(temp_sum)

    return reward_to_go

```

```
[220]: class Node:
    def __init__(self, state, player_type=0):
        # 3x3 matrix representing the board state
        self.state = np.copy(state)

        # to store a list of (action, child_node) tuples
        self.children = []

        self.player_type = player_type
        self.simulator = Simulator()
        self.reward = self.simulator.get_reward(self.state)
        self.values = 0

    def is_terminal(self):
        if not self.children:
            return True
        else:
            return False
```

```
[226]: class TicTacToe:
    def __init__(self, root_state):
        self.root = Node(root_state, ENEMY)
        self.simulator = Simulator()
        self.state_list = []
        self.node_list = []

    def is_node_present(self, root_node, node_state):
        if np.any(node_state - self.root.state) == False:
            print("This should be printed once")
            return False, None

        child_list = root_node.children

        if len(child_list) != 0:
            for child in child_list:
                child_node = child[1]
                if np.any(node_state - child_node.state) == False:
                    return True, child_node
                else:
                    if_present, return_node = self.is_node_present(child_node, ↵
                                                     node_state)
                    if if_present == True:
```

```

        return True, return_node

    return False, None

def build_tree(self, node, player_type, agent_action=None):
    if node is None:
        node = self.root

    if self.simulator.has_game_end(np.copy(node.state)) == True:
        return

    if player_type == ENEMY:
        init_Q_state = np.copy(node.state)

        if self.simulator.has_empty_cells(init_Q_state) == True:
            empty_cells = self.simulator.find_empty_cells(init_Q_state)
            num_empty_cells = len(empty_cells)

            for i in range(num_empty_cells):
                has_enemy_moved, modified_board_state = self.simulator.
                ↪enemy_move(np.copy(init_Q_state), empty_cells[i])
                assert(np.sum(modified_board_state) == -1)

                if has_enemy_moved == True:
                    present, child_node = self.is_node_present(self.root, ↪
                    ↪np.copy(modified_board_state))
                    ↪
                    present = False

                if present:
                    node.children.append((agent_action, child_node))
                else:
                    global num_tree_nodes
                    num_tree_nodes += 1
                    if (num_tree_nodes % 100 == 0):
                        print(num_tree_nodes)

                    child_node = Node(modified_board_state)
                    node.children.append((agent_action, child_node))

                    self.state_list.append(np.
                    ↪copy(modified_board_state))
                    self.node_list.append(child_node)

                    if self.simulator.
                    ↪has_game_end(modified_board_state) == True:
                        pass
                    else:

```

```

        self.build_tree(child_node, player_type=AGENT)

    elif player_type == AGENT:
        init_board_state = np.copy(node.state)

        if self.simulator.has_empty_cells(init_board_state) == True:
            empty_cells = self.simulator.find_empty_cells(init_board_state)
            num_empty_cells = len(empty_cells)

            for i in range(num_empty_cells):
                # agent will take an action to get to Q-state
                agent_action = empty_cells[i]
                has_agent_moved, modified_Q_state = self.simulator.
→agent_move(np.copy(init_board_state), agent_action)
                assert(np.sum(modified_Q_state) == 0)

                if has_agent_moved == True:

                    if self.simulator.has_game_end(modified_Q_state) ==
→True:
                        if self.simulator.has_agent_won(modified_Q_state) ==
→== True:
                            child_node = Node(modified_Q_state)
                            node.children.append((agent_action, child_node))
                        else:
                            node.state = np.copy(modified_Q_state)
                            self.build_tree(node, player_type=ENEMY,
→agent_action=agent_action)
                            node.state = np.copy(init_board_state)

    def print_trajectory(self):
        traj = []
        continue_loop = True
        tree_node = self.root

        while continue_loop:
            traj.append(tree_node)
            node_state = tree_node.state

            if tree_node.is_terminal() == False:
                child_list = tree_node.children
                num_child = len(child_list)

                child_element = child_list[np.random.randint(0, num_child)]

                tree_node = child_element[1]
            else:

```

```

        if (self.simulator.has_agent_won(tree_node.state) == True):
            print("Agent Won !!!")
        elif (self.simulator.has_enemy_won(tree_node.state) == True):
            print("Enemy Won !!!")
        elif (self.simulator.has_game_drawn(tree_node.state) == True):
            print("Game Drawn !!!")
        else:
            print("Something is wrong")

    continue_loop = False

    reward_to_go = self.simulator.compute_reward_to_go(traj)
    traj_len = len(traj)
    for j in range(traj_len):
        print(traj[j].state, ", R(s_i) = ", global_sim_obj.
→get_reward(traj[j].state), " G(s_i) = ", round(reward_to_go[j], 2), "\n")

```

[227]: AGENT, ENEMY = 0, 1  
game\_obj = TicTacToe(np.zeros((3,3)))

[228]: num\_tree\_nodes = 0  
game\_obj.build\_tree(None, player\_type=ENEMY)

100  
200  
300  
400  
500  
600  
700  
800  
900  
1000  
1100  
1200  
1300  
1400  
1500  
1600  
1700  
1800  
1900  
2000  
2100  
2200  
2300  
2400

```
2500  
2600  
2700
```

```
[192]: print("Total number of states in the tree: ", len(game_obj.state_list))  
print("Total number of nodes in the tree: ", len(game_obj.node_list))  
  
total_states = len(game_obj.state_list)  
unique_state_list = []  
unique_node_list = []  
  
for i in range(total_states):  
    include = True  
    total_unique_states = len(unique_state_list)  
  
    if (i % 1000 == 0):  
        print(i, '/', total_states, ',', total_unique_states)  
  
    for j in range(total_unique_states):  
        if np.any(game_obj.state_list[i] - unique_state_list[j]) == False:  
            include = False  
            break  
  
    if include:  
        unique_state_list.append(game_obj.state_list[i])  
        unique_node_list.append(game_obj.node_list[i])
```

```
Total number of states in the tree: 291681  
Total number of nodes in the tree: 291681  
0 / 291681 , 0  
1000 / 291681 , 197  
2000 / 291681 , 293  
3000 / 291681 , 329  
4000 / 291681 , 337  
5000 / 291681 , 449  
6000 / 291681 , 532  
7000 / 291681 , 558  
8000 / 291681 , 574  
9000 / 291681 , 654  
10000 / 291681 , 716  
11000 / 291681 , 738  
12000 / 291681 , 748  
13000 / 291681 , 752  
14000 / 291681 , 844  
15000 / 291681 , 868  
16000 / 291681 , 878  
17000 / 291681 , 882  
18000 / 291681 , 938
```

19000 / 291681 , 958  
20000 / 291681 , 963  
21000 / 291681 , 975  
22000 / 291681 , 991  
23000 / 291681 , 997  
24000 / 291681 , 1000  
25000 / 291681 , 1007  
26000 / 291681 , 1016  
27000 / 291681 , 1021  
28000 / 291681 , 1023  
29000 / 291681 , 1025  
30000 / 291681 , 1027  
31000 / 291681 , 1028  
32000 / 291681 , 1029  
33000 / 291681 , 1216  
34000 / 291681 , 1295  
35000 / 291681 , 1342  
36000 / 291681 , 1353  
37000 / 291681 , 1353  
38000 / 291681 , 1409  
39000 / 291681 , 1450  
40000 / 291681 , 1466  
41000 / 291681 , 1471  
42000 / 291681 , 1515  
43000 / 291681 , 1544  
44000 / 291681 , 1558  
45000 / 291681 , 1564  
46000 / 291681 , 1595  
47000 / 291681 , 1620  
48000 / 291681 , 1631  
49000 / 291681 , 1635  
50000 / 291681 , 1652  
51000 / 291681 , 1671  
52000 / 291681 , 1680  
53000 / 291681 , 1683  
54000 / 291681 , 1690  
55000 / 291681 , 1700  
56000 / 291681 , 1708  
57000 / 291681 , 1709  
58000 / 291681 , 1715  
59000 / 291681 , 1719  
60000 / 291681 , 1724  
61000 / 291681 , 1725  
62000 / 291681 , 1725  
63000 / 291681 , 1727  
64000 / 291681 , 1729  
65000 / 291681 , 1731  
66000 / 291681 , 1789

67000 / 291681 , 1876  
68000 / 291681 , 1930  
69000 / 291681 , 1936  
70000 / 291681 , 1937  
71000 / 291681 , 2008  
72000 / 291681 , 2045  
73000 / 291681 , 2057  
74000 / 291681 , 2058  
75000 / 291681 , 2070  
76000 / 291681 , 2095  
77000 / 291681 , 2102  
78000 / 291681 , 2102  
79000 / 291681 , 2122  
80000 / 291681 , 2134  
81000 / 291681 , 2137  
82000 / 291681 , 2137  
83000 / 291681 , 2151  
84000 / 291681 , 2160  
85000 / 291681 , 2163  
86000 / 291681 , 2163  
87000 / 291681 , 2164  
88000 / 291681 , 2176  
89000 / 291681 , 2179  
90000 / 291681 , 2180  
91000 / 291681 , 2181  
92000 / 291681 , 2188  
93000 / 291681 , 2191  
94000 / 291681 , 2191  
95000 / 291681 , 2193  
96000 / 291681 , 2195  
97000 / 291681 , 2196  
98000 / 291681 , 2197  
99000 / 291681 , 2235  
100000 / 291681 , 2288  
101000 / 291681 , 2301  
102000 / 291681 , 2301  
103000 / 291681 , 2301  
104000 / 291681 , 2357  
105000 / 291681 , 2374  
106000 / 291681 , 2376  
107000 / 291681 , 2378  
108000 / 291681 , 2412  
109000 / 291681 , 2419  
110000 / 291681 , 2419  
111000 / 291681 , 2420  
112000 / 291681 , 2432  
113000 / 291681 , 2435  
114000 / 291681 , 2435

115000 / 291681 , 2435  
116000 / 291681 , 2438  
117000 / 291681 , 2446  
118000 / 291681 , 2448  
119000 / 291681 , 2448  
120000 / 291681 , 2448  
121000 / 291681 , 2454  
122000 / 291681 , 2457  
123000 / 291681 , 2458  
124000 / 291681 , 2458  
125000 / 291681 , 2459  
126000 / 291681 , 2464  
127000 / 291681 , 2465  
128000 / 291681 , 2465  
129000 / 291681 , 2467  
130000 / 291681 , 2469  
131000 / 291681 , 2470  
132000 / 291681 , 2470  
133000 / 291681 , 2516  
134000 / 291681 , 2524  
135000 / 291681 , 2524  
136000 / 291681 , 2532  
137000 / 291681 , 2559  
138000 / 291681 , 2564  
139000 / 291681 , 2564  
140000 / 291681 , 2572  
141000 / 291681 , 2586  
142000 / 291681 , 2590  
143000 / 291681 , 2590  
144000 / 291681 , 2593  
145000 / 291681 , 2603  
146000 / 291681 , 2606  
147000 / 291681 , 2606  
148000 / 291681 , 2606  
149000 / 291681 , 2610  
150000 / 291681 , 2612  
151000 / 291681 , 2612  
152000 / 291681 , 2612  
153000 / 291681 , 2616  
154000 / 291681 , 2617  
155000 / 291681 , 2617  
156000 / 291681 , 2618  
157000 / 291681 , 2621  
158000 / 291681 , 2621  
159000 / 291681 , 2621  
160000 / 291681 , 2622  
161000 / 291681 , 2624  
162000 / 291681 , 2625

163000 / 291681 , 2625  
164000 / 291681 , 2644  
165000 / 291681 , 2648  
166000 / 291681 , 2648  
167000 / 291681 , 2648  
168000 / 291681 , 2663  
169000 / 291681 , 2666  
170000 / 291681 , 2666  
171000 / 291681 , 2666  
172000 / 291681 , 2673  
173000 / 291681 , 2679  
174000 / 291681 , 2679  
175000 / 291681 , 2679  
176000 / 291681 , 2679  
177000 / 291681 , 2683  
178000 / 291681 , 2688  
179000 / 291681 , 2688  
180000 / 291681 , 2688  
181000 / 291681 , 2689  
182000 / 291681 , 2694  
183000 / 291681 , 2694  
184000 / 291681 , 2694  
185000 / 291681 , 2695  
186000 / 291681 , 2696  
187000 / 291681 , 2696  
188000 / 291681 , 2696  
189000 / 291681 , 2697  
190000 / 291681 , 2698  
191000 / 291681 , 2698  
192000 / 291681 , 2698  
193000 / 291681 , 2699  
194000 / 291681 , 2700  
195000 / 291681 , 2701  
196000 / 291681 , 2701  
197000 / 291681 , 2701  
198000 / 291681 , 2708  
199000 / 291681 , 2708  
200000 / 291681 , 2708  
201000 / 291681 , 2709  
202000 / 291681 , 2714  
203000 / 291681 , 2714  
204000 / 291681 , 2714  
205000 / 291681 , 2714  
206000 / 291681 , 2719  
207000 / 291681 , 2719  
208000 / 291681 , 2719  
209000 / 291681 , 2719  
210000 / 291681 , 2720

211000 / 291681 , 2723  
212000 / 291681 , 2723  
213000 / 291681 , 2723  
214000 / 291681 , 2725  
215000 / 291681 , 2726  
216000 / 291681 , 2726  
217000 / 291681 , 2726  
218000 / 291681 , 2728  
219000 / 291681 , 2728  
220000 / 291681 , 2728  
221000 / 291681 , 2728  
222000 / 291681 , 2729  
223000 / 291681 , 2729  
224000 / 291681 , 2729  
225000 / 291681 , 2729  
226000 / 291681 , 2730  
227000 / 291681 , 2731  
228000 / 291681 , 2731  
229000 / 291681 , 2731  
230000 / 291681 , 2732  
231000 / 291681 , 2732  
232000 / 291681 , 2732  
233000 / 291681 , 2732  
234000 / 291681 , 2732  
235000 / 291681 , 2733  
236000 / 291681 , 2733  
237000 / 291681 , 2733  
238000 / 291681 , 2733  
239000 / 291681 , 2734  
240000 / 291681 , 2734  
241000 / 291681 , 2734  
242000 / 291681 , 2734  
243000 / 291681 , 2735  
244000 / 291681 , 2735  
245000 / 291681 , 2735  
246000 / 291681 , 2735  
247000 / 291681 , 2736  
248000 / 291681 , 2736  
249000 / 291681 , 2736  
250000 / 291681 , 2737  
251000 / 291681 , 2737  
252000 / 291681 , 2737  
253000 / 291681 , 2737  
254000 / 291681 , 2737  
255000 / 291681 , 2738  
256000 / 291681 , 2738  
257000 / 291681 , 2738  
258000 / 291681 , 2738

```
259000 / 291681 , 2738
260000 / 291681 , 2739
261000 / 291681 , 2739
262000 / 291681 , 2739
263000 / 291681 , 2739
264000 / 291681 , 2739
265000 / 291681 , 2739
266000 / 291681 , 2739
267000 / 291681 , 2739
268000 / 291681 , 2739
269000 / 291681 , 2739
270000 / 291681 , 2739
271000 / 291681 , 2739
272000 / 291681 , 2739
273000 / 291681 , 2739
274000 / 291681 , 2739
275000 / 291681 , 2739
276000 / 291681 , 2739
277000 / 291681 , 2739
278000 / 291681 , 2739
279000 / 291681 , 2739
280000 / 291681 , 2739
281000 / 291681 , 2739
282000 / 291681 , 2739
283000 / 291681 , 2739
284000 / 291681 , 2739
285000 / 291681 , 2739
286000 / 291681 , 2739
287000 / 291681 , 2739
288000 / 291681 , 2739
289000 / 291681 , 2739
290000 / 291681 , 2739
291000 / 291681 , 2739
```

```
[229]: print("Total number of states in the tree: ", len(game_obj.state_list))
print("Total number of nodes in the tree: ", len(game_obj.node_list))
```

```
Total number of states in the tree: 2739
Total number of nodes in the tree: 2739
```

### 0.0.2 Question 2: 5 Random Full Trajectories from some initial state to a terminal state with arbitrary actions in each step

```
[33]: gamma = 0.9

# 5 random trajectories
for itr in range(5):
```

```

print("Trajectory -", itr+1)
board = np.zeros((3,3), dtype=np.int32)
game_obj.print_trajectory()
print("\n")

```

```

Trajectory - 1
Game Draw !!!
[[ 0  0  0]
 [ 0  0  0]
 [-1  0  0]] R(s_i) = 1 G(s_i) = 3.439

[[ 0 -1  0]
 [ 0  0  0]
 [-1  1  0]] R(s_i) = 1 G(s_i) = 2.71

[[ 0 -1  0]
 [ 0  1 -1]
 [-1  1  0]] R(s_i) = 1 G(s_i) = 1.9

[[ 1 -1  0]
 [ 0  1 -1]
 [-1  1 -1]] R(s_i) = 1 G(s_i) = 1.0

[[ 1 -1  1]
 [-1  1 -1]
 [-1  1 -1]] R(s_i) = 0 G(s_i) = 0.0

Trajectory - 2
Game Draw !!!
[[ 0  0 -1]
 [ 0  0  0]
 [ 0  0  0]] R(s_i) = 1 G(s_i) = 3.439

[[ 0  0 -1]
 [ 0  1  0]
 [-1  0  0]] R(s_i) = 1 G(s_i) = 2.71

[[ 0  1 -1]
 [ 0  1  0]
 [-1 -1  0]] R(s_i) = 1 G(s_i) = 1.9

[[ 0  1 -1]
 [ 1  1 -1]
 [-1 -1  0]] R(s_i) = 1 G(s_i) = 1.0

[[ -1  1 -1]
 [ 1  1 -1]
 [ -1 -1  0]]

```

```

[-1 -1  1]]    R(s_i) =  0  G(s_i) =  0.0

Trajectory - 3
Enemy Won !!!
[[ 0  0  0]
 [ 0  0 -1]
 [-1  0  0]]  R(s_i) =  1  G(s_i) = -3.122

[[ 0  1  0]
 [ 0  0 -1]
 [-1  0  0]]  R(s_i) =  1  G(s_i) = -4.58

[[ 0  1  0]
 [ 0 -1 -1]
 [-1  1  0]]  R(s_i) =  1  G(s_i) = -6.2

[[ -1  1  0]
 [ 0 -1 -1]
 [-1  1  1]]  R(s_i) =  1  G(s_i) = -8.0

[[ -1  1 -1]
 [ 1 -1 -1]
 [-1  1  1]]  R(s_i) = -10  G(s_i) = -10.0

Trajectory - 4
Game Draw !!!
[[ -1  0  0]
 [ 0  0  0]
 [ 0  0  0]]  R(s_i) =  1  G(s_i) =  3.439

[[ -1  1  0]
 [ 0  0 -1]
 [ 0  0  0]]  R(s_i) =  1  G(s_i) =  2.71

[[ -1  1  0]
 [ 1 -1 -1]
 [ 0  0  0]]  R(s_i) =  1  G(s_i) =  1.9

[[ -1  1 -1]
 [ 1 -1 -1]
 [ 1  0  0]]  R(s_i) =  1  G(s_i) =  1.0

[[ -1  1 -1]
 [ 1 -1 -1]
 [ 1 -1  1]]  R(s_i) =  0  G(s_i) =  0.0

Trajectory - 5
Enemy Won !!!

```

```

[[ 0  0 -1]
 [ 0  0  0]
 [ 0  0  0]] R(s_i) =  1  G(s_i) = -4.58

[[ 0  0 -1]
 [ 0 -1  0]
 [ 0  0  1]] R(s_i) =  1  G(s_i) = -6.2

[[ 0 -1 -1]
 [ 1 -1  0]
 [ 0  0  1]] R(s_i) =  1  G(s_i) = -8.0

[[ 0 -1 -1]
 [ 1 -1  0]
 [ 1 -1  1]] R(s_i) = -10  G(s_i) = -10.0

```

```
[210]: # show 5 random trajectory
for i in range(5):
    print("Trajectory -", (i+1))
    game_obj.print_trajectory()
    print("\n")
```

```

Trajectory - 1
Agent Won !!!
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]] 10.0

[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [-1.  0.  0.]] 10.0

[[ 0.  0.  0.]
 [-1.  0.  0.]
 [-1.  0.  1.]] 10.0

[[ 0.  0.  1.]
 [-1.  0.  0.]
 [-1. -1.  1.]] 10.00000000000002

[[ 1.  0.  1.]
 [-1. -1.  0.]
 [-1. -1.  1.]] 10.0

[[ 1.  1.  1.]
 [-1. -1.  0.]
 [-1. -1.  1.]] 10.0

```

```

Trajectory - 2
Enemy Won !!!
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]] -1.8098

[[ 0. 0. 0.]
 [ 0. 0. -1.]
 [ 0. 0. 0.]] -3.122

[[ 0. 0. 1.]
 [ 0. 0. -1.]
 [ 0. -1. 0.]] -4.580000000000001

[[-1. 0. 1.]
 [ 0. 1. -1.]
 [ 0. -1. 0.]] -6.200000000000001

[[-1. 0. 1.]
 [ 0. 1. -1.]
 [-1. -1. 1.]] -8.0

[[-1. 1. 1.]
 [-1. 1. -1.]
 [-1. -1. 1.]] -10.0

```

```

Trajectory - 3
Enemy Won !!!
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]] -4.580000000000001

[[ 0. 0. 0.]
 [ 0. 0. 0.]
 [ 0. -1. 0.]] -6.200000000000001

[[ 0. -1. 0.]
 [ 0. 0. 0.]
 [ 0. -1. 1.]] -8.0

[[ 0. -1. 0.]
 [ 0. -1. 0.]
 [ 1. -1. 1.]] -10.0

```

```

Trajectory - 4
Agent Won !!!
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]] 10.0

[[ 0. 0. -1.]
 [ 0. 0. 0.]
 [ 0. 0. 0.]] 10.0

[[ 0. 0. -1.]
 [ 0. 1. -1.]
 [ 0. 0. 0.]] 10.0

[[ 0. 0. -1.]
 [-1. 1. -1.]
 [ 1. 0. 0.]] 10.000000000000002

[[ -1. 1. -1.]
 [-1. 1. -1.]
 [ 1. 0. 0.]] 10.0

[[ -1. 1. -1.]
 [-1. 1. -1.]
 [ 1. 1. 0.]] 10.0

```

```

Trajectory - 5
Agent Won !!!
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]] 10.0

[[ 0. 0. 0.]
 [-1. 0. 0.]
 [ 0. 0. 0.]] 10.0

[[ 1. 0. 0.]
 [-1. 0. -1.]
 [ 0. 0. 0.]] 10.0

[[ 1. 0. 0.]
 [-1. 0. -1.]
 [ 1. -1. 0.]] 10.000000000000002

```

```
[[ 1. -1.  0.]
 [-1.  1. -1.]
 [ 1. -1.  0.]] 10.0
```

```
[[ 1. -1.  0.]
 [-1.  1. -1.]
 [ 1. -1.  1.]] 10.0
```

```
[212]: num_unique_states = len(unique_state_list)
for i in range(num_unique_states):
    assert(not np.any(unique_state_list[i] - unique_node_list[i].state))
```

### 0.0.3 Value Iteration Method

```
[284]: def value_iteration(state_list, node_list, gamma=0.9, epsilon=1e-1):
    delta_thresh = epsilon * (1-gamma) / gamma

    num_states = len(state_list)
    new_values_list = np.zeros((num_states, 1))
    curr_values_list = np.zeros((num_states, 1))

    has_converged = False

    itr_count = 0
    while has_converged == False:
        curr_values_list = np.copy(new_values_list)
        delta = 0

        for i in range(num_states):
            board_state = state_list[i]
            state_reward = game_obj.simulator.get_reward(board_state)
            child_list = node_list[i].children
            num_child = len(child_list)
            assert(np.any(board_state - node_list[i].state) == False)
#
#                print("Assertion successful !!!")
#
#                print("State: ", board_state)
#                print("Num child: ", num_child)

            if num_child > 0:
                child_dict = {}

                for j in range(num_child):
```

```

        child_obj = child_list[j]      # (action, child node)
        child_parent_action = child_obj[0]
        x,y = child_parent_action

        if (x,y) not in child_dict:
            child_dict[(x,y)] = []
            print("X & Y: ", x, y)

        child_node = child_obj[1]
#        print("Child State:", child_node.state)
        child_dict[(x,y)].append(child_node)

        actions = list(child_dict.keys())
        num_actions = len(actions)
#        print("Actions: ", actions, "Num Actions: ", num_actions)

        prob = {}
        for j in range(num_actions):
            action = actions[j]
            x, y = action
            num_child_per_action = len(child_dict[(x,y)])
            prob[(x,y)] = 1 / num_child_per_action

#        print("Probability: ", prob)

        max_expected_val = -math.inf
        for j in range(num_actions):
            action = actions[j]
            x,y = action
            pot_states_list = child_dict[(x,y)]
            num_pot_states = len(pot_states_list)

            expected_val = 0
            for k in range(num_pot_states):
                pot_state = pot_states_list[k].state
                # search the index of this state in the state list

                pot_idx = None
                for itr in range(num_states):
#                    print("This is USA: ", state_list[itr])
#                    print("This is India: ", pot_state)
                    if (np.any(state_list[itr] - pot_state) == False):
                        pot_idx = itr
                        break

                expected_val = expected_val + prob[(x,y)] * ↵
                curr_values_list[pot_idx,0]

```

```

        if expected_val > max_expected_val:
            max_expected_val = expected_val

        new_values_list[i,0] = state_reward + gamma * max_expected_val
    else:
        new_values_list[i,0] = state_reward

delta = np.max(np.abs(new_values_list - curr_values_list))

if (delta < delta_thresh):
    print("CONVERGED !!!")
    has_converged = True

itr_count += 1
print("Iteration -", itr_count, " Delta: ", delta)

return new_values_list

```

#### 0.0.4 Convergence of Value Iteration

[285]: value\_iteration(game\_obj.state\_list, game\_obj.node\_list)  
*# value\_iteration(unique\_state\_list, unique\_node\_list)*

```

Iteration - 1  Delta:  10.0
Iteration - 2  Delta:  9.0
Iteration - 3  Delta:  2.7
Iteration - 4  Delta:  0.7290000000000001
Iteration - 5  Delta:  0.6561000000000003
Iteration - 6  Delta:  0.59049
Iteration - 7  Delta:  0.531441
Iteration - 8  Delta:  0.4464104400000002
Iteration - 9  Delta:  0.4017693960000006
Iteration - 10  Delta:  0.36159245640000126
Iteration - 11  Delta:  0.3167992684337131
Iteration - 12  Delta:  0.2736143762443506
Iteration - 13  Delta:  0.24602750733455903
Iteration - 14  Delta:  0.21908693170321225
Iteration - 15  Delta:  0.1908121615032483
Iteration - 16  Delta:  0.1673971159904335
Iteration - 17  Delta:  0.15002438820403885
Iteration - 18  Delta:  0.13243632172606734
Iteration - 19  Delta:  0.11567225805046455
Iteration - 20  Delta:  0.10233589525054221
Iteration - 21  Delta:  0.09116882026603257
Iteration - 22  Delta:  0.08014547370492053

```

```
Iteration - 23 Delta: 0.07035403218834091
Iteration - 24 Delta: 0.062413647624214974
Iteration - 25 Delta: 0.055311922652578005
Iteration - 26 Delta: 0.04859675011159226
Iteration - 27 Delta: 0.0428369520591545
Iteration - 28 Delta: 0.03798666029622666
Iteration - 29 Delta: 0.03355022092487481
Iteration - 30 Delta: 0.029514978296555583
Iteration - 31 Delta: 0.026078217803782167
Iteration - 32 Delta: 0.02308882050015093
Iteration - 33 Delta: 0.02036062899932567
Iteration - 34 Delta: 0.017942396845153752
Iteration - 35 Delta: 0.01586593770478295
Iteration - 36 Delta: 0.014025133102466114
Iteration - 37 Delta: 0.012364554259098881
CONVERGED !!!
Iteration - 38 Delta: 0.01091078916686783
```

```
[285]: array([[ 8.70912631,
   [ 8.57304523],
   [-1.4       ],
   ...,
   [ 7.17461998],
   [ 9.01393594],
   [ 8.70912631]])
```

## Question 5

Initial game state

s <sub>0</sub>		
X		X
O	O	
X		

Reward function

$$R(s) = \begin{cases} +10 & \text{player wins} \\ -10 & \text{player loses} \\ 0 & \text{draw} \\ c & \text{else} \end{cases}$$

Come up with 2 different q & a

$$q_1 < 0, q_2 > 0$$

$$V^{x^*}(s_0) = c + \gamma \max_{a \in A(s_0)} \sum_{s'} P(s'|s_0, a) V^{x^*}(s')$$

$$= c + \gamma \max [A_1, A_2, A_3, A_4]$$

$$\text{where } A_1 = \sum_{s'} P(s'|s_0, a_{11}=1) V^{x^*}(s')$$

$$A_2 = \sum_{s'} P(s'|s_0, a_{12}=1) V^{x^*}(s')$$

$$A_3 = \sum_{s'} P(s'|s_0, a_{21}=1) V^{x^*}(s')$$

$$A_4 = \sum_{s'} P(s'|s_0, a_{22}=1) V^{x^*}(s')$$

$$A_1 = \frac{1}{3} \left[ c + \gamma \max_{a \in A(s_1)} \{ P(s'_1|s_1, a_{11}=1) \cdot 10, P(s''_1|s_1, a_{12}=1) \cdot 0 \} \right]$$

$$+ \frac{1}{3} \left[ c + \gamma \max_{a \in A(s_2)} \{ P(s'_2|s_2, a_{11}=1) \cdot 10, P(s''_2|s_2, a_{12}=1) \cdot 0 \} \right]$$

$$+ \frac{1}{3} \left[ c + \gamma \max_{a \in A(s_3)} \{ P(s'_3|s_3, a_{11}=1) \cdot 10, P(s''_3|s_3, a_{12}=1) \cdot 10 \} \right]$$

$$= \frac{1}{3} \left[ c + \gamma \max (1 \times 10, 1 \times 0) \right] + \frac{1}{3} \left[ c + \gamma \max (1 \times 10, 1 \times 0) \right]$$

$$+ \frac{1}{3} \left[ c + \gamma \max (1 \times 10, 1 \times 10) \right] \cdot$$

$$= \frac{1}{3} (c + 10\gamma) + \frac{1}{3} (c + 10\gamma) + \frac{1}{3} (c + 10\gamma) = (c + 10\gamma)$$

X = -1
O = +1
empty = 0

$$A_2 = \frac{1}{3} \times 10 + \frac{1}{3} \times 10 + \frac{1}{3} \times 10 = 10$$

$$\begin{aligned} A_3 &= \frac{1}{3}[-10] + \frac{1}{3}[c + \gamma \max(1 \times 10, 1 \times -10)] + \frac{1}{3}[c + \gamma \max(1 \times 10, 1 \times 10)] \\ &= -\frac{10}{3} + \frac{1}{3}(c + 10\gamma) + \frac{1}{3}(c + 10\gamma) \\ &= \frac{2}{3}(c + 10\gamma) - \frac{10}{3} \end{aligned}$$

$$\begin{aligned} A_4 &= \frac{1}{3}[-10] + \frac{1}{3}[c + \gamma \max(1 \times 0, 1 \times -10)] + \frac{1}{3}[c + \gamma \max(1 \times 10, 1 \times 0)] \\ &= -\frac{10}{3} + \frac{1}{3}c + \frac{1}{3}(c + 10\gamma) = \frac{2c}{3} + \frac{10\gamma}{3} - \frac{10}{3} \end{aligned}$$

Now  $\pi^*(s_0) = c + \gamma \max(A_1, A_2, A_3, A_4)$

$$= c + \gamma \max \left[ c + 10\gamma, 10, \frac{2c}{3} + 20\gamma - \frac{10}{3}, \frac{2c}{3} + 10\gamma - \frac{10}{3} \right]$$

= Using  $\gamma = 0.9$

$$\pi^*(s_0) = c + \frac{9}{10} \max \left[ c + 9, 10, \frac{2c}{3} + \frac{8}{3}, \frac{2c}{3} - \frac{1}{3} \right]$$

Let  $c_1 = -1 < 0$  and  $c_2 = 2 > 0$

$$\pi^*(s_0) \Big|_{c=c_1} = c_1 + \frac{9}{10} \max [8, 10, 2, -1]$$



Corresponds to 2nd action  $a_{12}=1$  at  $s_0$ .

$$\pi^*(s_0) \Big|_{c=c_2} = c_2 + \frac{9}{10} \max [11, 10, 4, 1]$$

Corresponds to 1st action  $a_{01}=1$  at  $s_0$ .

Choosing  $c_2 = -1$  and  $c_2 = 2$  achieves the desired purpose.. and optimal decision/action changes from  $a_{12}=1$  to  $a_{01}=1$

Under  $c_1 = -1$

Optimal action at  $s_0$

$$a_{12}=1$$

x	x	x
0	0	0
x		

Under  $c_2 = 2$

Optimal action at  $s_0$   $a_{01}=1$

x	o	x
0	0	
x		

Now,  $V^{x^*}(s_0) = c + \max_{a \in A(s_0)} \sum_{s'} p(s'|s_0, a) V^{x^*}(s')$

$$\therefore V^{x^*}(s_0) = \max_{a \in A(s_0)} \left[ c + \gamma \sum_{s'} p(s'|s_0, a) V^{x^*}(s') \right]$$

$Q(s_0, a)$

Under  $c=c_1=-1$

$$Q(s_0, a_{01}=1) = c_1 + \gamma \cdot 8 = 6.2$$

$$Q(s_0, a_{12}=1) = c_1 + \gamma \cdot 10 = 8$$

$$Q(s_0, a_{21}=1) = c_1 + \gamma \cdot \left( \frac{2c_1}{3} + \frac{8}{3} \right) = 0.8$$

$$Q(s_0, a_{22}=1) = c_1 + \gamma \left( \frac{2c_1}{3} - \frac{1}{3} \right) = -1.9$$

Under  $c=c_2=2$

$$Q(s_0, a_{01}=1) = c_2 + \gamma \cdot (c_2 + q) = 11.9$$

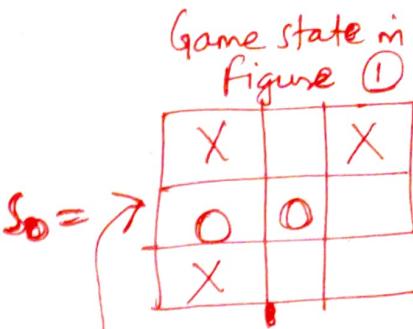
$$Q(s_0, a_{12}=1) = c_2 + \gamma \cdot 10 = 11$$

$$Q(s_0, a_{21}=1) = c_2 + \gamma \left( \frac{2c_2}{3} + \frac{8}{3} \right) = 5.6$$

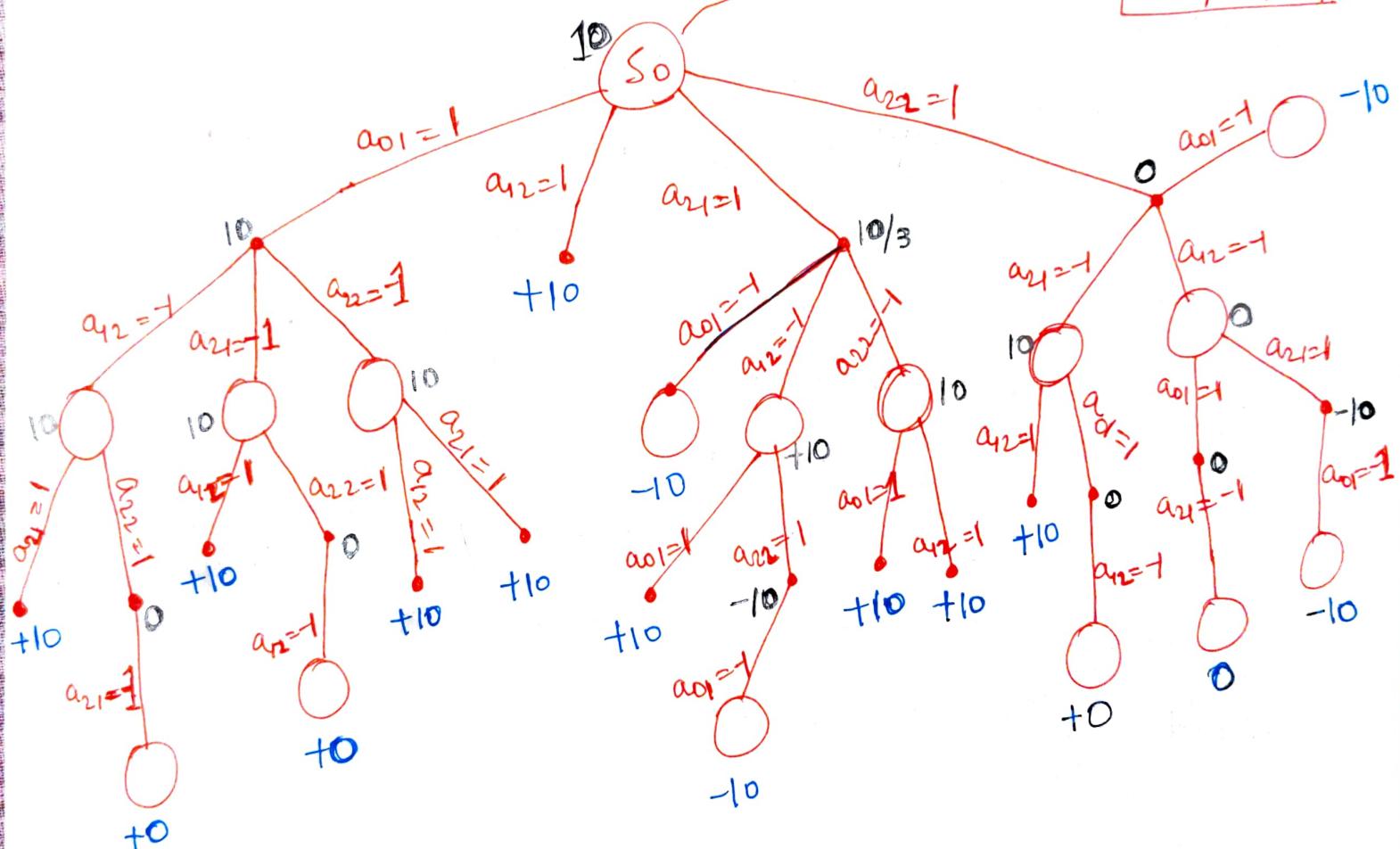
$$Q(s_0, a_{22}=1) = c_2 + \gamma \left( \frac{2c_2}{3} - \frac{1}{3} \right) = 2.9$$

## Question 6

$$a_{ij} = \begin{cases} +1 & \text{represents } O \text{ (circle)} \\ -1 & \text{represents } X \text{ (cross)} \\ 0 & \text{represents empty cell} \end{cases}$$



-1	0	-1
1	1	0
-1	0	0



→ Using Expectimax, the player at  $S_0$  sees 4 options with respective scores - 10, 10,  $\frac{10}{3}$ , 0. Therefore, the max player at  $S_0$  chooses the path with highest expected score = +10.

→ Optimal Action at  $S_0$  = Either of  $a_{01}=1$  or  $a_{12}=1$

In string format representation,

Optimal Actions are  $a_{01} = 1$  (or  $a = 1$ )  
and  $a_{12} = 1$  (or  $a = 5$ )

Reason for change of ~~reward~~ optimal action based on reward definition

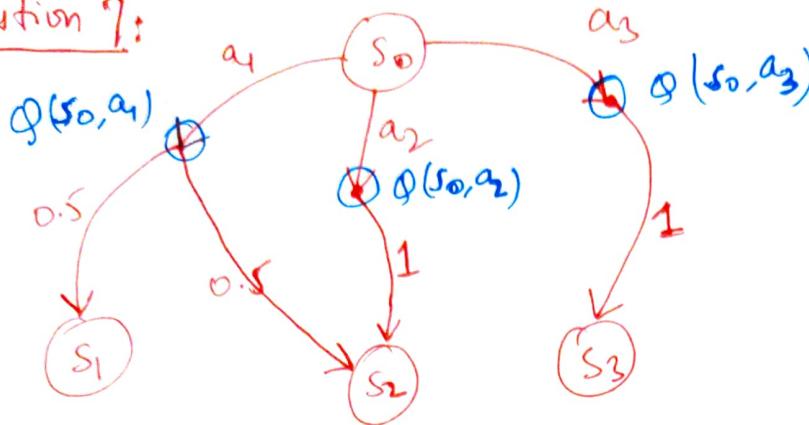
When we assign a low reward value to states, the optimal action should be to finish the game as soon as possible and get to the terminal state. On the other hand, when reward values are high, the optimal actions ~~may~~ shift towards collecting more non-terminal high reward values before finishing the game.

For example, in Question 5, when  $G=1$ , optimal action is  $a_{02}=1$  at  $s_0$  in which case the player wins immediately. But when  $G=2$ , the optimal action changes to  $a_{01}=1$  at  $s_0$ , which does not guarantee a win (either future win or draw), but more expected discounted reward sum compared to the action  $a_{12}=1$ .

## ② Function Approximation in RL

$$\gamma = 0.5$$

Question 7:



$$s_i = (i+1) \quad a_i = i$$

$$P(s, a) = \alpha s + \beta a$$

$$Q(s_0, a_1) = \alpha + \beta \quad \text{--- } ①$$

$$Q(s_0, a_2) = \alpha + 2\beta \quad \text{--- } ②$$

$$Q(s_0, a_3) = \alpha + 3\beta \quad \text{--- } ③$$

$s_1, s_2, s_3$  - terminal states

$$P(s_1 | s_0, a_1) = 0.5 \quad P(s_2 | s_0, a_2) = 1$$

$$P(s_3 | s_0, a_3) = 0.5 \quad P(s_3 | s_0, a_3) = 1$$

Let  $R(s_0) = R_0, R(s_1) = R_1 \quad \} \text{ unknowns.}$

Let  $R(s_2) = R_2, R(s_3) = R_3 \quad \} \text{ unknowns.}$

$$Q(s_0, a_1) = R_0 + \gamma \sum_{s_1} P(s_1 | s_0, a_1) V(s_1) = R_0 + \gamma [P(s_1 | s_0, a_1) V(s_1) + P(s_2 | s_0, a_1) V(s_2)]$$

$$= R_0 + \gamma \left[ \frac{1}{2} V(s_1) + \frac{1}{2} V(s_2) \right] = R_0 + \frac{1}{4} [V(s_1) + V(s_2)]$$

Since  $s_1, s_2, s_3$  are terminal states,  $V(s_1) = R_1, V(s_2) = R_2, V(s_3) = R_3$

$$Q(s_0, a_1) = R_0 + \frac{1}{4} [R_1 + R_2] \quad \text{--- } ④$$

$$Q(s_0, a_2) = R_0 + \gamma \cdot P(s_2 | s_0, a_2) V(s_2) = R_0 + \frac{1}{2} R_2 \quad \text{--- } ⑤$$

$$Q(s_0, a_3) = R_0 + \gamma P(s_3 | s_0, a_3) V(s_3) = R_0 + \frac{1}{2} R_3. \quad \text{--- } ⑥$$

From ①-④, ②-⑤, ③-⑥ (In order for linear Q-function to correctly represent Q-values at  $s_0$ )

$$\alpha + \beta = R_0 + \frac{R_1 + R_2}{4}, \quad \alpha + 2\beta = R_0 + \frac{R_2}{2}, \quad \alpha + 3\beta = R_0 + \frac{R_3}{2}.$$

We have 3 equations and 4 unknowns ( $R_0, R_1, R_2, R_3$ ).

$$\beta = \frac{R_2 - R_1}{4} = \frac{R_3 - R_2}{2}$$

$$\Rightarrow R_2 = (4\beta + R_1)$$

$$= (4\beta + 2\alpha - 2R_0)$$

$$R_2 = 2(\alpha + 2\beta - R_0)$$

$$\alpha = R_0 + \frac{R_1}{2}$$

$$\Rightarrow R_1 = 2(\alpha - R_0)$$

$$R_3 = 2\beta + R_2$$

$$= 2\beta + 2\alpha + 4\beta - 2R_0$$

$$R_3 = 2(\alpha + 3\beta - R_0)$$

$$R(s_0) = R_0$$

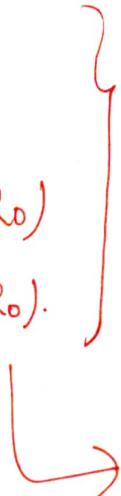
$$R(s_1) = 2(\alpha - R_0)$$

$$R(s_2) = 2(\alpha + 2\beta - R_0)$$

$$R(s_3) = 2(\alpha + 3\beta - R_0)$$

where  $\alpha, \beta \in \mathbb{R}$  are weight parameters  
that can be arbitrarily chosen.

Let  $\alpha = 2$  and  $\beta = 3$ . and  $R_0 = 1$ .



Q-value function

$R(s_0) = 1$	$R(s_2) = 14$
$R(s_1) = 2$	$R(s_3) = 20$

$$Q(s_0, a_1) = \alpha + \beta = R_0 + \frac{1}{4}[R_1 + R_2] = 5$$

$$Q(s_0, a_2) = \alpha + 2\beta = R_0 + \frac{R_2}{2} = 8$$

$$Q(s_0, a_3) = \alpha + 3\beta = R_0 + \frac{R_3}{2} = 11$$

The above reward function  
correctly represents  
all Q-values by the linear  
function  $Q(s, a) = \alpha s + \beta a$   
at state  $s_0$ .

② Goal: To design a reward function s.t.  $Q(s,a) = \alpha s + \beta a$  does not work.

Question  
7.2

$$Q(s,a) = \alpha s + \beta a \quad \text{---} \quad ①$$

→ Cannot represent all Q-values at  $s_0$  for any  $\alpha, \beta \in \mathbb{R}$

	$Q^L(s,a)$ <u>Linear function</u>
$Q(s_0, a_1)$	$\alpha + \beta$
$Q(s_0, a_2)$	$\alpha + 2\beta$
$Q(s_0, a_3)$	$\alpha + 3\beta$

$$\begin{matrix} Q^T(s,a) \\ \text{True Value} \end{matrix}$$

$$\frac{R_0 + \frac{R_1 + R_2}{2}}{4}$$

$$\frac{R_0 + \frac{R_2}{2}}{2}$$

$$\frac{R_0 + \frac{R_3}{2}}{2}$$

$Q^T(s,a)$ : True value computed using reward

$Q^L(s,a)$ : computed using linear function

Let reward function be  $R(s) = s$

$$\text{Thus } R(s_0) = 1 \quad R(s_1) = 2 \quad R(s_2) = 3 \quad R(s_3) = 4.$$

Then

$$\begin{cases} \cancel{\Rightarrow} Q^L(s_0, a_1) = \alpha + \beta = \frac{1}{4} = Q^T(s_0, a_1) \\ \Rightarrow Q^L(s_0, a_2) = \alpha + 2\beta = \frac{2}{2} = Q^T(s_0, a_2) \\ \Rightarrow Q^L(s_0, a_3) = \alpha + 3\beta = 3 = Q^T(s_0, a_3) \end{cases} \quad \left. \begin{array}{l} 3 \text{ eqns.} \\ 2 \text{ unknowns} \end{array} \right\}$$

$$\Rightarrow (\alpha, \beta) = \left( \frac{3}{2}, -\frac{1}{2} \right) \Rightarrow \frac{3}{2} + \frac{1}{2} = 2 \neq \frac{1}{4}. \quad \text{(does not satisfy 1st equation)}$$

$$(\alpha, \beta) = \left( 2, -\frac{1}{4} \right) \Rightarrow 2 + \frac{3}{4} = \frac{11}{4} \neq 3$$

→ Thus, for  $R(s) = s$ , the above set of linear equations does not have any solution.

→ For any choice of  $(\alpha, \beta)$ , the linear function  $Q^L(s,a) = \alpha s + \beta a$  does not represent all the true values of  $Q^T(s,a)$ , if  $R(s) = s$ .

$$Q^L(s_0, a_1) = \alpha + \beta$$

$$Q^L(s_0, a_2) = \alpha + 2\beta$$

$$Q^L(s_0, a_3) = \alpha + 3\beta$$

$$Q^T(s_0, a_1) = \frac{9}{4} \quad \text{--- } ①$$

$$Q^T(s_0, a_2) = \frac{5}{2} \quad \text{--- } ②$$

$$Q^T(s_0, a_3) = 3 \quad \text{--- } ③$$

Solving ① & ②

$$\alpha + \beta = \frac{9}{4}$$

$$\alpha + 2\beta = \frac{5}{2}$$

↓

$$\alpha = 2, \beta = \frac{1}{4}$$

↓  
putting into the ③ equation:

$$\alpha + 3\beta = \frac{11}{4} \neq 3 = Q^T(s_0, a_3)$$

Similarly solving ① & ③,  $\alpha = \frac{15}{8}, \beta = \frac{3}{8}$  and putting into ② eqn.

$$\alpha + 2\beta = \frac{21}{8} \neq \frac{5}{2} = Q^T(s_0, a_2)$$

Solving ② & ③

$$\alpha + 2\beta = \frac{5}{2}$$

$$\alpha + 3\beta = 3$$

↓

$$\alpha = \frac{3}{2}, \beta = \frac{1}{2}$$

↓  
putting into the ① equation..

$$\alpha + \beta = 2 \neq \frac{9}{4} = Q^T(s_0, a_1)$$

∴ Hence for  $R(s) = 5$ , the above set of linear equations ①, ② & ③ does not have any solution. That is, the function  $Q(s, a) = \alpha s + \beta a$  cannot represent all true  $Q$ -values at  $s_0$  for any choice of  $\alpha$  &  $\beta$ .

7.3

Reward function for previous question

$$R(s) = s$$

$$R(s_0) = R_0 = 1$$

$$R(s_1) = R_1 = 2$$

$$R(s_2) = R_2 = 3$$

$$R(s_3) = R_3 = 4$$

True Value

$$Q^T(s, a)$$

$$Q^T(s_0, a_1) = R_0 + \frac{R_1 + R_2}{4} = \frac{9}{4}$$

$$Q^T(s_0, a_2) = R_0 + \frac{R_2}{2} = \frac{5}{2} = \frac{10}{4}$$

$$Q^T(s_0, a_3) = R_0 + \frac{R_3}{2} = 3 = \frac{12}{4}$$

Goal: Design a function approximation  $Q^*(s, a)$  s.t all Q-values at  $s_0$  can be correctly represented.

Let 
$$Q(s, a) = \frac{9}{4}s + \frac{1}{4}f(a)$$

s.t.  $Q(s_0, a_1) = \frac{9}{4} + \frac{1}{4}f(a_1) = \frac{9}{4} \Rightarrow f(a_1) = 0$

$$Q(s_0, a_2) = \frac{9}{4} + \frac{1}{4}f(a_2) = \frac{10}{4} \Rightarrow f(a_2) = 1$$

$$Q(s_0, a_3) = \frac{9}{4} + \frac{1}{4}f(a_3) = \frac{12}{4} \Rightarrow f(a_3) = 3$$

Need to design  $f(a)$  s.t.  $f(1) = 0$   $f(2) = 1$   $f(3) = 3$

~~Let~~  ~~$f(a) = pa + qa + r$~~

such that 
$$\begin{cases} f(1) = p + q + r = 0 \\ f(2) = 4p + 2q + r = 1 \\ f(3) = 9p + 3q + r = 3 \end{cases}$$

Unique solution for  $p, q, r$ .

$$p = \frac{1}{2}, q = -\frac{1}{2}, r = 0$$

$$f(a) = \frac{1}{2}a^2 + qa + r$$

$$= \frac{1}{2}a^2 - \frac{1}{2}a = \frac{1}{2}a(a-1)$$

Thus,  $Q(s, a) = \frac{q}{4}s + \frac{1}{4}f(a)$

$$\boxed{Q(s, a) = \frac{q}{4}s + \frac{1}{8}a(a-1)}$$

For  $R(s) = s$ ,  $Q(s, a) = \alpha s + \beta a$  cannot correctly represent all  $Q$ -values at  $s_0$  but  $Q(s, a) = \frac{q}{4}s + \frac{1}{8}a(a-1)$  can correctly represent all  $Q$ -values at  $s_0$ .

# CSE257\_A3\_Blackjack

December 5, 2021

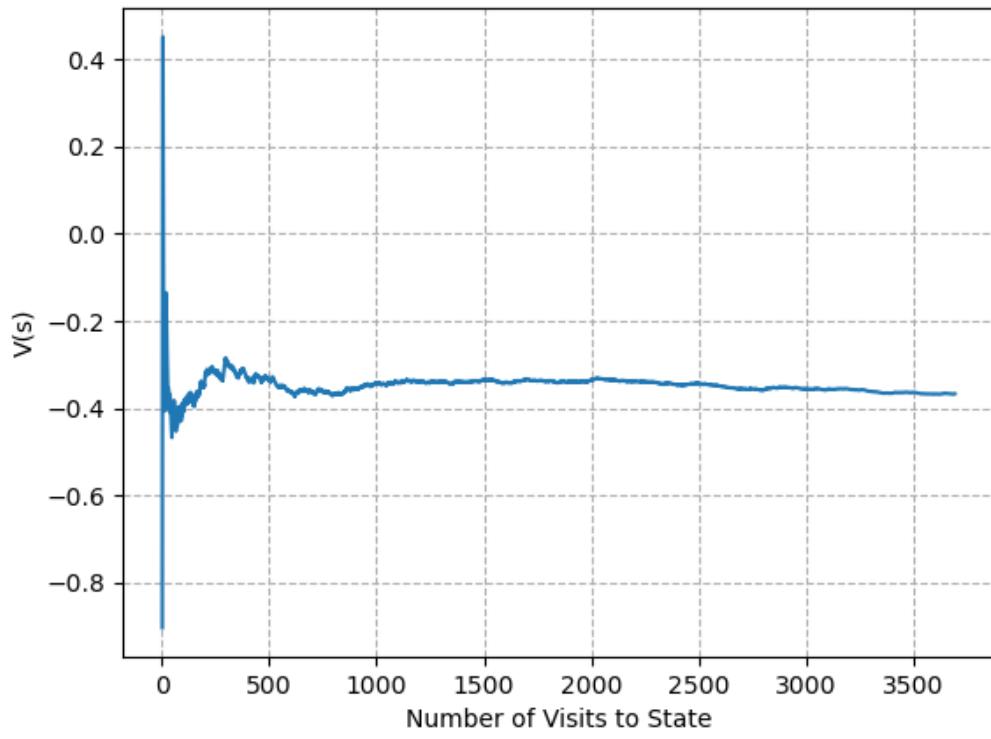
## 0.0.1 Question 8 (3 Extra Points):

Select two game states, A and B: In State A the player's sum of cards is 10, and in State B the sum of cards is 20. Plot how the value estimate of each state changes over the number of visits to the state until the values converge, under Monte Carlo policy evaluation and Temporal-Difference policy evaluation, respectively; so 4 plots in total.

```
(base) siriusA@Barkat-MacAir : ~/Desktop/UCSD/courses/Fall_2021/CSE257/assignments/assignment3/blackjack-main
$ python3 main.py -t 3
  MC 1000000/1000000
++++ PASSED MC with 0 wrong values
  TD 1000000/1000000
++++ PASSED TD with 0 wrong values
  Q 1000000/1000000
++++ PASSED Q-Learning with 0 wrong values
```

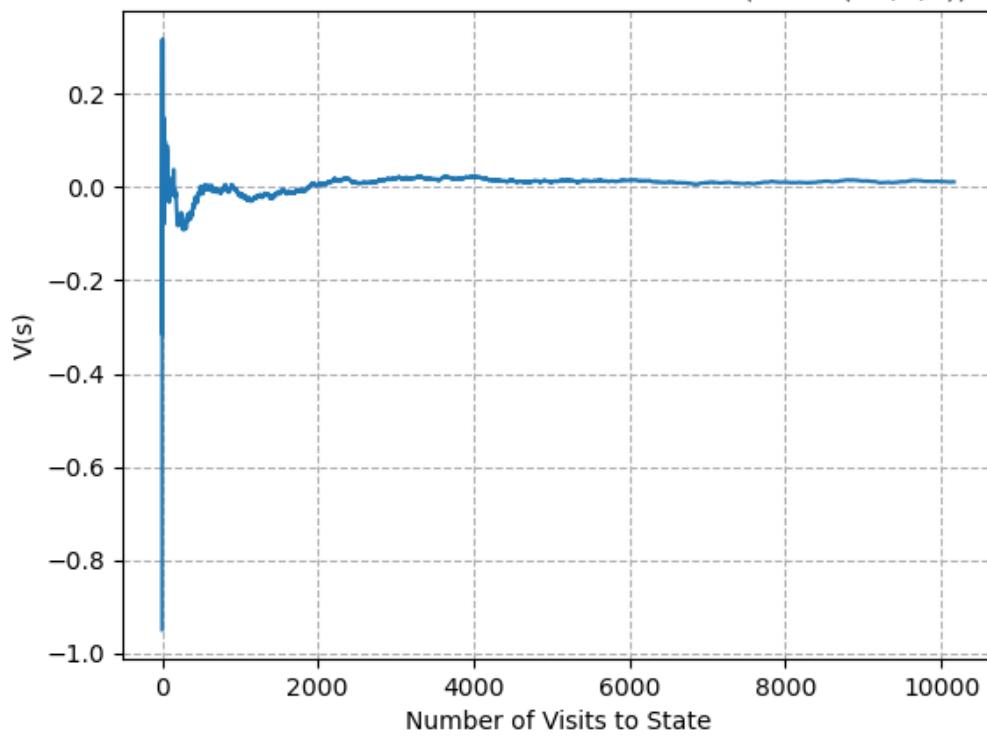
The Monte-Carlo values over the number of visits are stable whereas the Temporal-Difference state values are more fluctuating as compared to Monte-Carlo. This is because, in Monte-Carlo, each update to the state values happens after taking the expectation over large number of simulation sequences. On the other hand, in Temporal difference, each update happens with every incoming

MC: Value Estimate VS Number of Visits ( $s = A (10,0,1)$ )

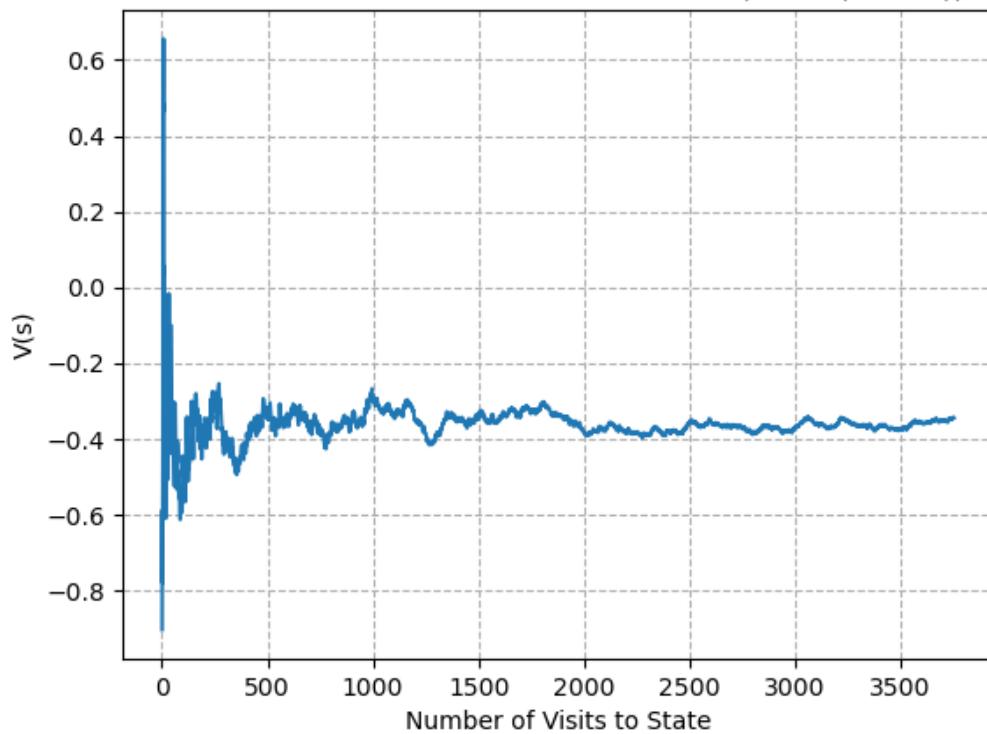


sample.

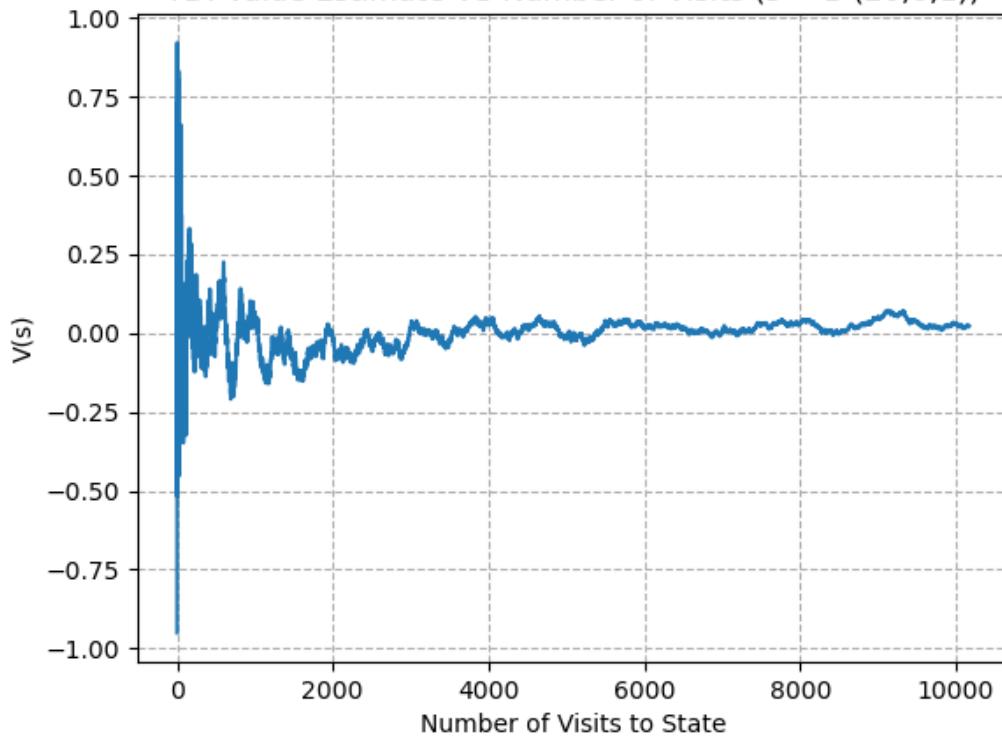
MC: Value Estimate VS Number of Visits ( $s = B (20,0,1)$ )



TD: Value Estimate VS Number of Visits ( $s = A (10,0,1)$ )

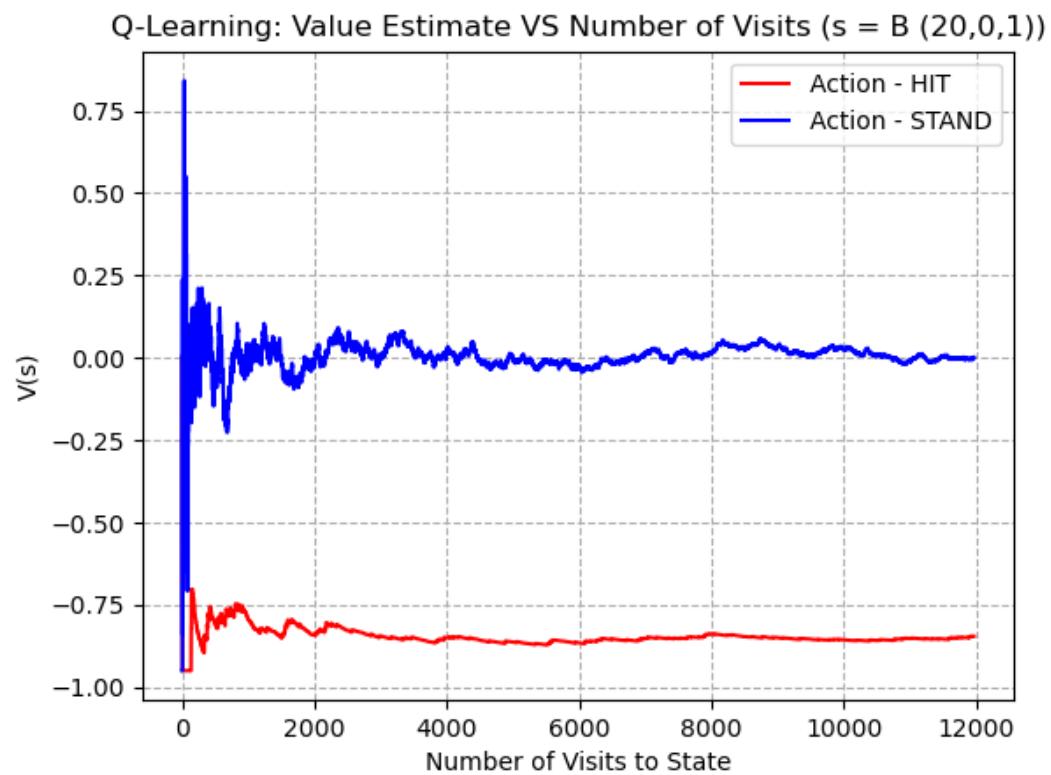
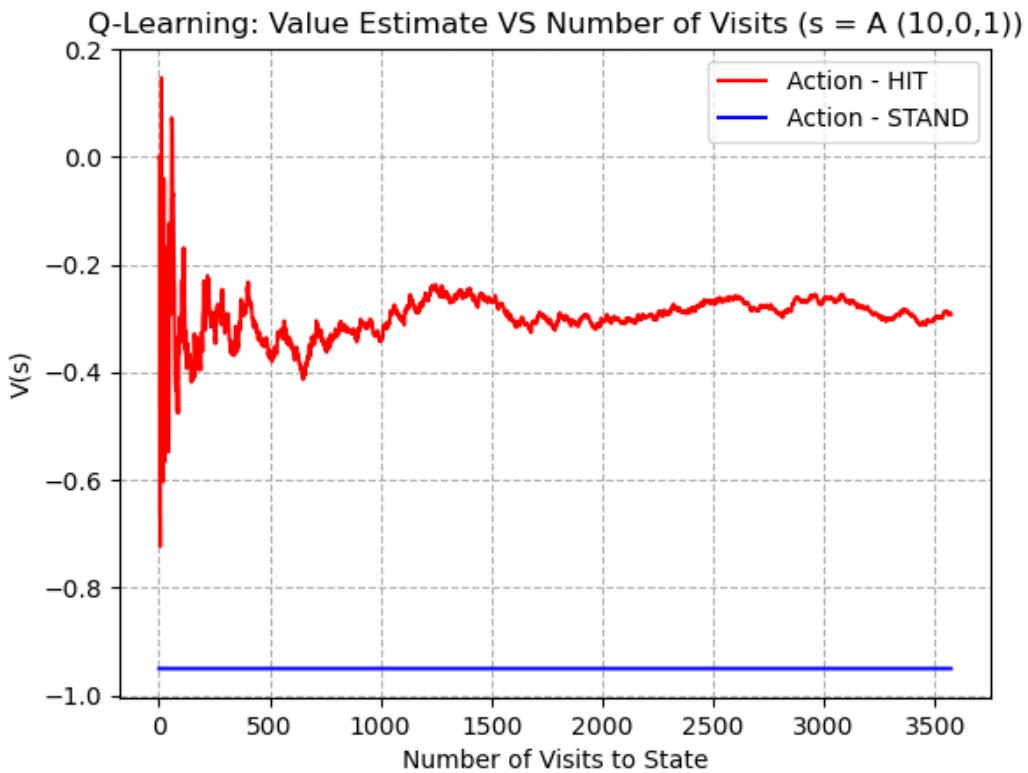


TD: Value Estimate VS Number of Visits ( $s = B (20,0,1)$ )



### **0.0.2 Question 9 (3 Extra Points):**

Perform Q-learning and plot how the Q-value changes over the number of visits to each action for the same two game states you selected above, until you have run Q-learning for long enough so that the Q-value converges at least on some action for each state (note: a very bad action may receive a small number of visits, so this requirement is saying you only need to wait till the better action has been visited enough times so that the Q-value of it stabilizes).



Also plot the cumulative winning rate over the number of plays in the game: for every n number of plays (x-axis), show the ratio w/n (y-axis) where w is the total number of wins out of the n plays.

For plotting the cumulative win-rate vs number of game plays, I have set the `self.autoQL = True` and `self.autoPlay = True`. Then, I am running sufficiently large number of iterations (=5000). In each iteration, the Q-Learning algorithm (50 simulations) runs, which updates the Q-values of the states followed by the game playing function. The cumulative winning rate stabilizes at around ~ 41 %.

