CZ3005 2020 Fall Assignment 2: Reinforcement Learning

Bisakha Das:U1823460E

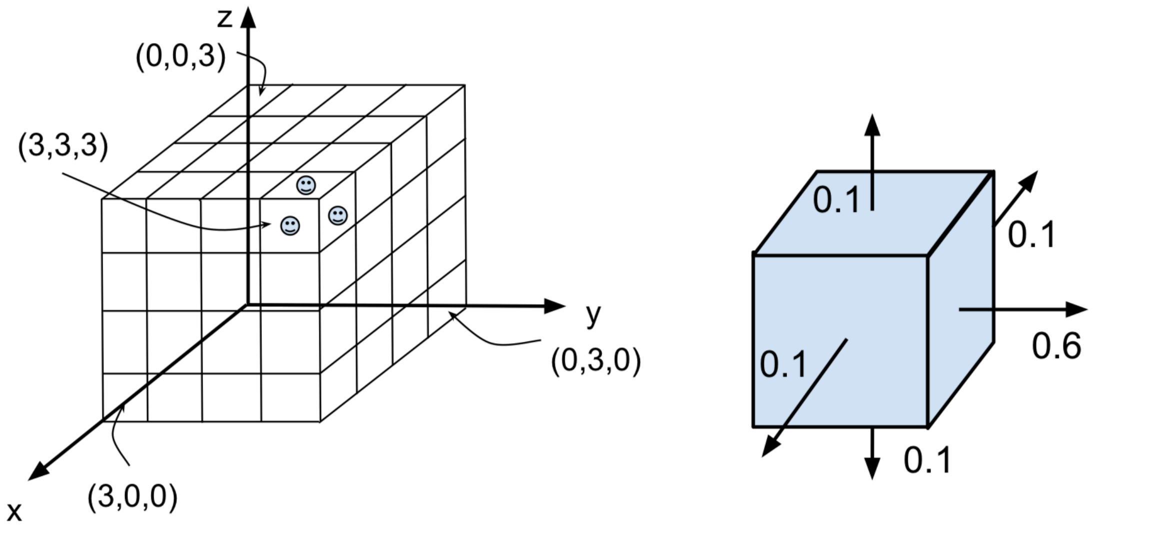
Table of Contents

Introduction

In this project, I chose to implement Q-learning for the given 3D grid-world-based environment. Q-learning is an off policy reinforcement learning algorithm, and given our stationary terminal state environment, my algorithm seeks to learn a policy which will maximise the total reward.

Problem Statement Definition (as per the assignment overview documentation)

- Environment: Treasure Hunting Grid World



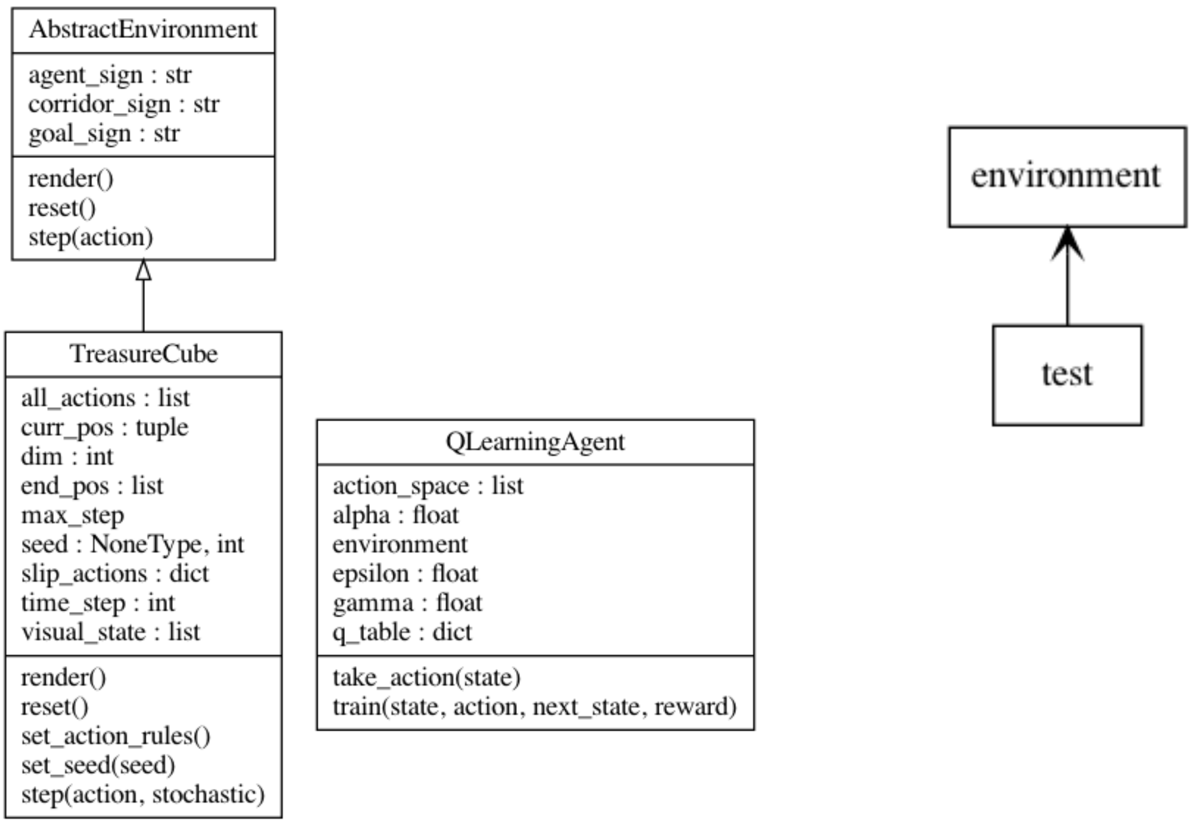
(a) 3D grid world. Smile faces represent terminal states which give reward 1. (b) The illustration of transition, e.g., the intended action is RIGHT

Figure 1: Illustration of treasure hunting in a cube

- State: a 3D coordinate, which indicates the current position where the agent is. The initial state is (0, 0, 0) and there is only one terminal state: (3,3,3).
- Action: The action space is (forward, backward, left, right, up, down). The agent needs to select one of them to navigate in the environment.
- Reward: The agent will receive 1 reward when it arrives at the terminal states, or otherwise receive -0.1 reward.
- Transition: The intended movement happens with probability 0.6. With probability 0.1, the agent ends up in one of the states perpendicular to the intended direction. If a collision with a wall happens, the agent stays in the same state.

Solution Algorithm

Class Dependencies



The above diagram shows the class dependencies and the respective variables and functions in each class. Although there were some changes made to the Environment file from the original, however for this report, I shall only go over the algorithmic changes made to the Test file.

Creating a q-table

From the environment illustration we can see that there are 64 (4^3) possible states and 6 actions for our q-table (forward, backward, left, right, up, down). Since the grid world is 3D, the states are in a (x,y,z) format, and the q-table itself follows the shape of [state, action], hence [64, 6]

We store all the q values in a dictionary of dictionaries, and all the values in the table are initialised to 0 in the beginning. We will later update and store episode q values and parse through the table in an attempt to select the best action based on the q-value.

The code is as follows:

```python
# INITIALISING THE Q_TABLE

self.q_table = dict() # Store all Q-values in dictionary of dictionaries

for x in range(4): # Loop through all possible grid spaces, create sub-dictionary for each
    for y in range(4):
        for z in range(4):
            self.q_table[(z,x,y)] = {'left':0,'right':0,'forward':0,'backward':0,'up':0,'down':0}
            #Initialise with zero values for possible moves
```

Choosing action based on the q-table

The agent interacts with the environment and decides it actions based on values in the q-table, then proceeds to update the table for future decisions.

The action taken by a q-learning agent is either exploration or exploitation, and the choice between these two types of actions depend on the epsilon value. If a random float draw from a uniform distribution from the numpy library is lower than the epsilon value, then the action selected will be exploration, else it will be exploitation.

Exploration allows the agent to act randomly and explore the environment, possibly discovering a new state that would have otherwise been unexplored during the exploitation stage. On the contrary, exploitation lets the agent select an action based on the maximum value in the q-table, promising maximum future reward. Furthermore, if there are multiple optimal actions, then the algorithm will chose a random action amongst those.

My epsilon value is 0.01, as given in the assignment overview documentation. Since epsilon indicates the ratio between exploration and exploitation, an epsilon value of 0.01 would indicate the ratio to be 1:99 (exploration : exploitation). Hence, a low value such as mine indicates that the algorithm favours exploitation over exploration.

The code for the function is as follows:

```python
def take_action(self, state):
    # Returns either the optimal action which will maximise the reward,
    # or a random exploratory action.
    # The above choice depends on the value of epsilon

    if np.random.uniform(0,1) < self.epsilon:
        action = self.action_space[np.random.randint(0, len(self.action_space))]

    else:
        qValues = self.q_table[self.environment.curr_pos]
        maxValue = max(qValues.values())
        action = np.random.choice([k for k, v in qValues.items() if v == maxValue])

    return action
```

Updating the q-table

The q-table is updated after each step till the agent reaches the terminal state, which in our case is (3,3,3).

The basic algorithm for updating is as follows:

```
Initialize Q-table
Repeat for each episode:
    Initialize state and episode reward array
    Repeat for each step in an episode:
        Choose an action using the policy derived from Q
        Take the action, observe the reward collected and the next state
        Use an update rule for Q and update the table
        Move to the next state
    Until terminal state is reached
```

The code for applying the above algorithm is as follows:

```
def test_cube(max_episode, max_step, q_plot, q_table_output, rendEnv_output, epiSummary):
    env = TreasureCube(max_step=max_step)
    agent = QLearningAgent(env)
    reward_per_episode = []
    totalSteps=0
    totalRewards=0

    for epsisode_num in range(0, max_episode):
        state = env.reset()
        terminate = False
        t=0
        episode_reward = 0
        while not terminate:
            state = env.curr_pos
            action = agent.take_action(state)
            reward, terminate, next_state = env.step(action)
            next_state = env.curr_pos

            if (rendEnv_output==True):
                env.render()
                print(f'step: {t}, action: {action}, reward: {reward}')
            t += 1

            #Updating the Q values
            agent.train(state, action, next_state, reward)
            state = next_state
            episode_reward += reward

        reward_per_episode.append(episode_reward)
        if epiSummary==True:
            print(f'epsisode: {epsisode_num}, total_steps: {t} episode reward: {episode_reward}')

        totalSteps+=t

    for r in range(len(reward_per_episode)):
        totalRewards += reward_per_episode[r]

    print("Average steps per episode=", totalSteps/(max_episode))
    print("Average rewards per episode=", totalRewards/(max_episode))
```

The test_cube function takes in many hyperparameters as we see above (namely, *max_episode*, *max_step*, *q_plot*, *q_table_output*, *rendEnv_output* and *epiSummary*). These are related with user inputs for generating desired output which I shall go over in more detail in the last section of this report.

Other than that, the code closely follows the algorithm mentioned above.

Function to update the q values:

```
def train(self, state, action, next_state, reward):
    #Updates the Q-value table using Q-learning

    qValues = self.q_table[next_state]
    max_q_value = max(qValues.values())
    current_q_value = self.q_table[state][action]

    self.q_table[state][action] = current_q_value +
                                  self.alpha * (reward + self.gamma * max_q_value - current_q_value)
```

The above function shows the update rule for Q, which is based on the Bellman Equation. The variables used (namely *alpha*, *reward* and *gamma*) are defined as follows:

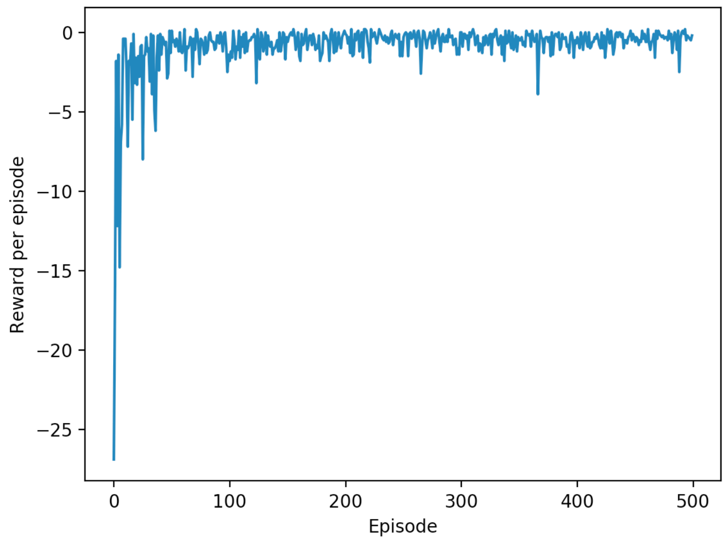
Alpha: Alpha is the learning rate, assigned a value of 0.5 as per the assignment overview documentation. Such a value means the ratio between the agent accepting the new value and acting upon the old values is perfectly balanced. As we can see in the update rule equation, the alpha value is multiplied with the temporal difference and added to the previous q-value, essentially meaning that the agent's consideration moves towards the new values.

Reward: Reward is the value gained after an action is completed. As defined in our problem statement, the agent will receive 1 reward when it arrives at the terminal states, and -0.1 otherwise.

Gamma: Gamma is the discount factor, assigned a value of 0.99 as per the assignment overview documentation. As we can see in the update rule equation, this value is multiplied with the maximum future reward, and therefore helps in balancing between the rewards of the current state and next state.

Evaluation and Visualisation

Training Graph



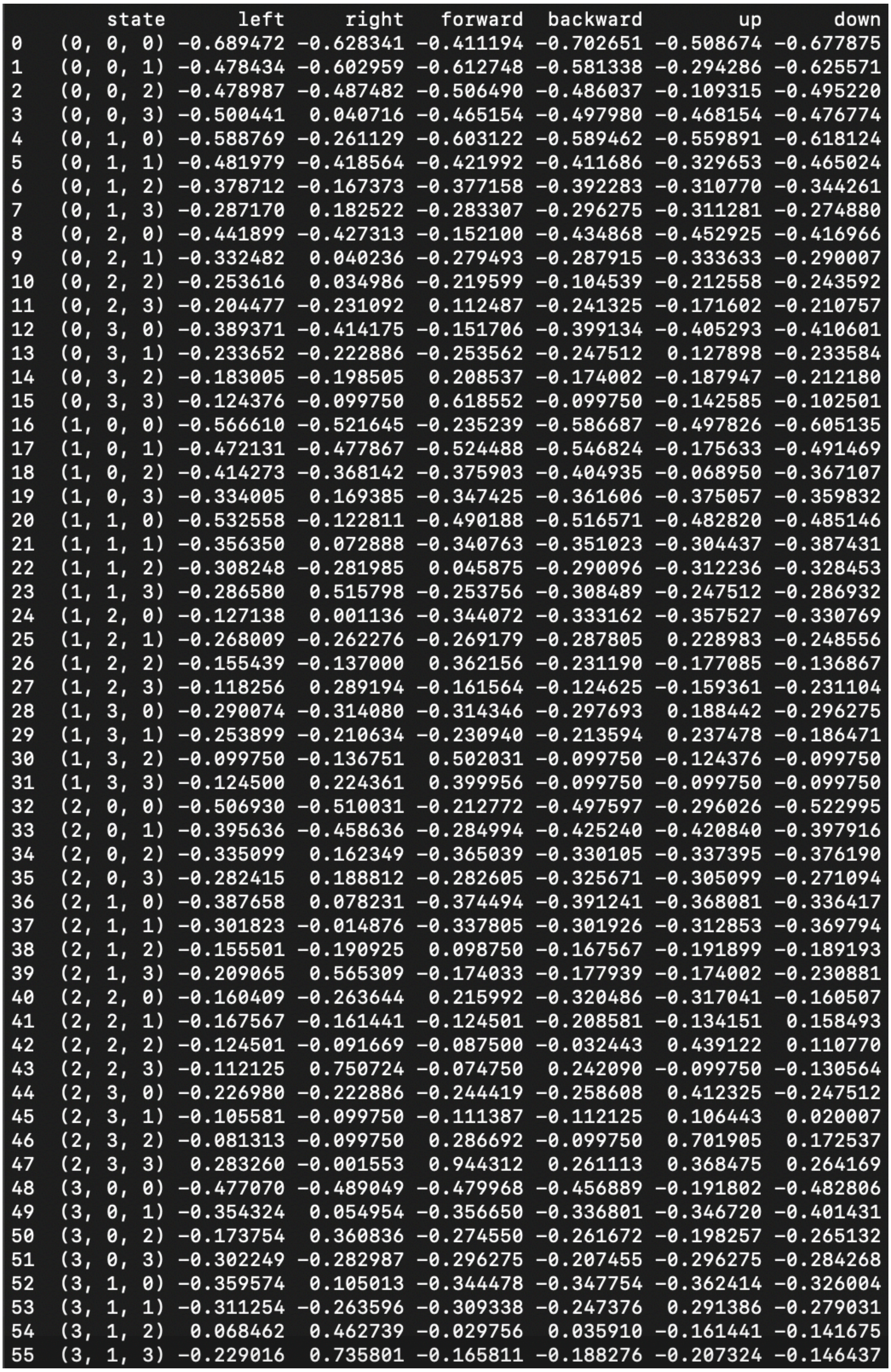
As we see from the above graph, q-learning significantly helps the agent learn. In the graph showing the learning progress of the q-learning, we see that the learning rate is significantly high and the values converge to achieve a reward maximised reward value.

Final Q table

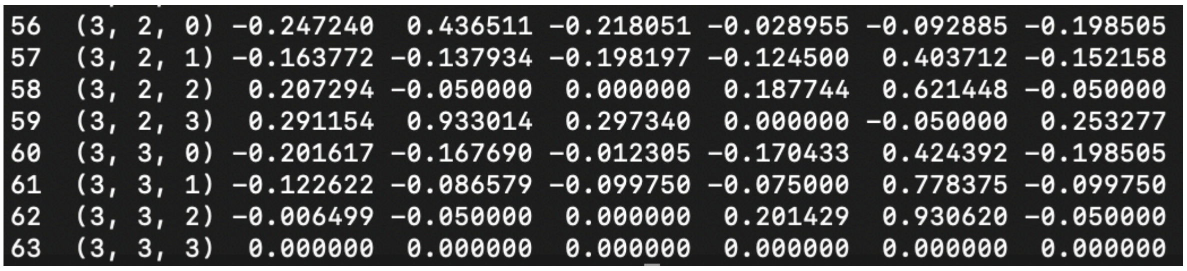
The q-table was in the format of a dictionary of dictionaries. The following function is used to print it out in the form of an array:

```
import csv
def qTableOutput(d):
    df = pd.DataFrame(d.items(), columns=["state", "action"])
    action = df["action"].apply(pd.Series)

    frames = [df, action]
    results = pd.concat(frames, axis=1)
    results = results.drop(results.columns[1], axis=1)

    with pd.option_context('display.max_rows', None, 'display.max_columns', None):
        print(results)

    # Uncomment the following line if you want to save a csv
    #pd.DataFrame(results).to_csv("qTable.csv")
```

Using the above function we can print out the q-values, as shown below:



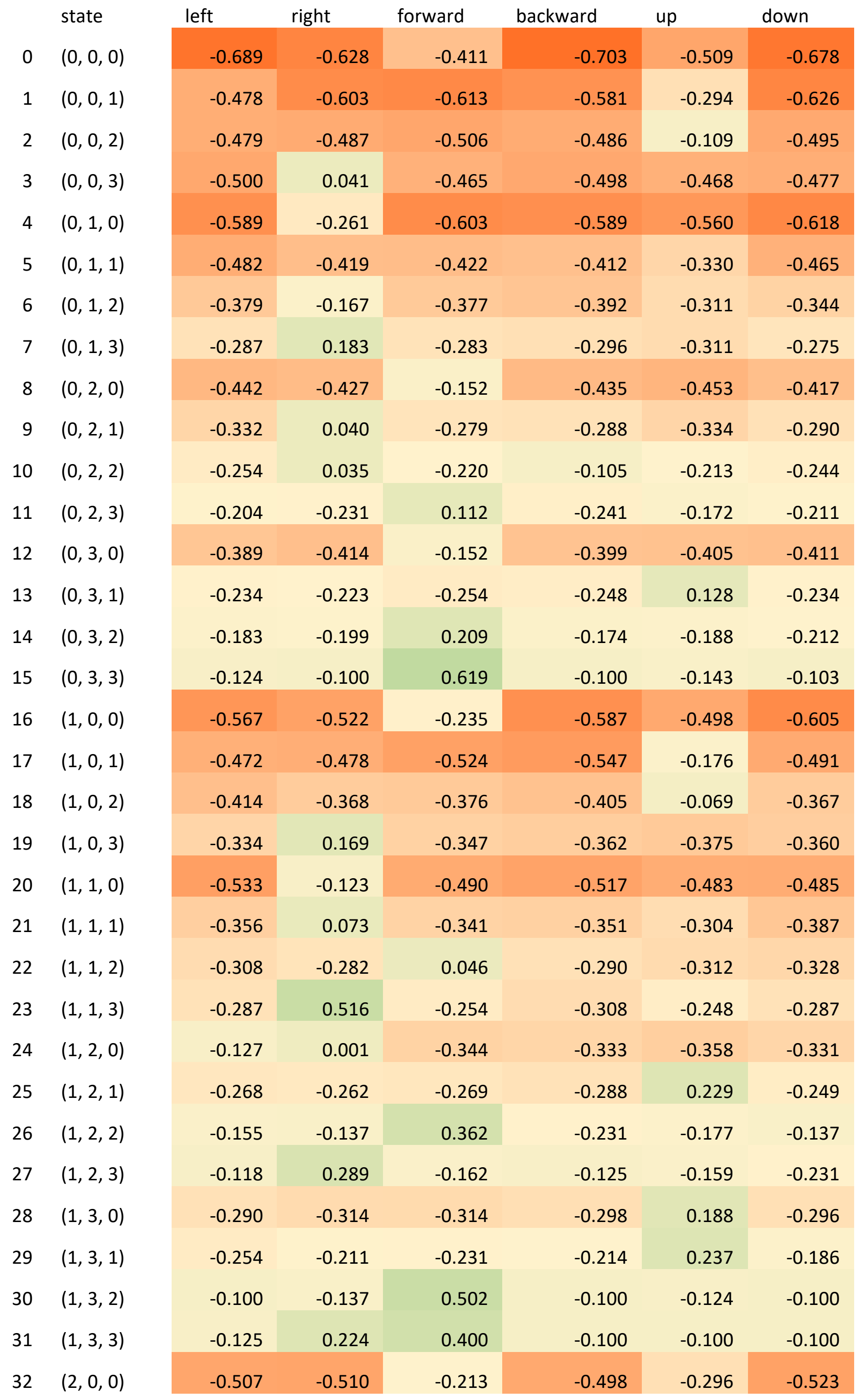
```
        state      left     right   forward  backward        up      down
0   (0, 0, 0) -0.689472 -0.628341 -0.411194 -0.702651 -0.508674 -0.677875
1   (0, 0, 1) -0.478434 -0.602959 -0.612748 -0.581338 -0.294286 -0.625571
2   (0, 0, 2) -0.478987 -0.487482 -0.506490 -0.486037 -0.109315 -0.495220
3   (0, 0, 3) -0.500441  0.040716 -0.465154 -0.497980 -0.468154 -0.476774
4   (0, 1, 0) -0.588769 -0.261129 -0.603122 -0.589462 -0.559891 -0.618124
5   (0, 1, 1) -0.481979 -0.418564 -0.421992 -0.411686 -0.329653 -0.465024
6   (0, 1, 2) -0.378712 -0.167373 -0.377158 -0.392283 -0.310770 -0.344261
7   (0, 1, 3) -0.287170  0.182522 -0.283307 -0.296275 -0.311281 -0.274880
8   (0, 2, 0) -0.441899 -0.427313 -0.152100 -0.434868 -0.452925 -0.416966
9   (0, 2, 1) -0.332482  0.040236 -0.279493 -0.287915 -0.333633 -0.290007
10  (0, 2, 2) -0.253616  0.034986 -0.219599 -0.104539 -0.212558 -0.243592
11  (0, 2, 3) -0.204477 -0.231092  0.112487 -0.241325 -0.171602 -0.210757
12  (0, 3, 0) -0.389371 -0.414175 -0.151706 -0.399134 -0.405293 -0.410601
13  (0, 3, 1) -0.233652 -0.222886 -0.253562 -0.247512  0.127898 -0.233584
14  (0, 3, 2) -0.183005 -0.198505  0.208537 -0.174002 -0.187947 -0.212180
15  (0, 3, 3) -0.124376 -0.099750  0.618552 -0.099750 -0.142585 -0.102501
16  (1, 0, 0) -0.566610 -0.521645 -0.235239 -0.586687 -0.497826 -0.605135
17  (1, 0, 1) -0.472131 -0.477867 -0.524488 -0.546824 -0.175633 -0.491469
18  (1, 0, 2) -0.414273 -0.368142 -0.375903 -0.404935 -0.068950 -0.367107
19  (1, 0, 3) -0.334005  0.169385 -0.347425 -0.361606 -0.375057 -0.359832
20  (1, 1, 0) -0.532558 -0.122811 -0.490188 -0.516571 -0.482820 -0.485146
21  (1, 1, 1) -0.356350  0.072888 -0.340763 -0.351023 -0.304437 -0.387431
22  (1, 1, 2) -0.308248 -0.281985  0.045875 -0.290096 -0.312236 -0.328453
23  (1, 1, 3) -0.286580  0.515798 -0.253756 -0.308489 -0.247512 -0.286932
24  (1, 2, 0) -0.127138  0.001136 -0.344072 -0.333162 -0.357527 -0.330769
25  (1, 2, 1) -0.268009 -0.262276 -0.269179 -0.287805  0.228983 -0.248556
26  (1, 2, 2) -0.155439 -0.137000  0.362156 -0.231190 -0.177085 -0.136867
27  (1, 2, 3) -0.118256  0.289194 -0.161564 -0.124625 -0.159361 -0.231104
28  (1, 3, 0) -0.290074 -0.314080 -0.314346 -0.297693  0.188442 -0.296275
29  (1, 3, 1) -0.253899 -0.210634 -0.230940 -0.213594  0.237478 -0.186471
30  (1, 3, 2) -0.099750 -0.136751  0.502031 -0.099750 -0.124376 -0.099750
31  (1, 3, 3) -0.124500  0.224361  0.399956 -0.099750 -0.099750 -0.099750
32  (2, 0, 0) -0.506930 -0.510031 -0.212772 -0.497597 -0.296026 -0.522995
33  (2, 0, 1) -0.395636 -0.458636 -0.284994 -0.425240 -0.420840 -0.397916
34  (2, 0, 2) -0.335099  0.162349 -0.365039 -0.330105 -0.337395 -0.376190
35  (2, 0, 3) -0.282415  0.188812 -0.282605 -0.325671 -0.305099 -0.271094
36  (2, 1, 0) -0.387658  0.078231 -0.374494 -0.391241 -0.368081 -0.336417
37  (2, 1, 1) -0.301823 -0.014876 -0.337805 -0.301926 -0.312853 -0.369794
38  (2, 1, 2) -0.155501 -0.190925  0.098750 -0.167567 -0.191899 -0.189193
39  (2, 1, 3) -0.209065  0.565309 -0.174033 -0.177939 -0.174002 -0.230881
40  (2, 2, 0) -0.160409 -0.263644  0.215992 -0.320486 -0.317041 -0.160507
41  (2, 2, 1) -0.167567 -0.161441 -0.124501 -0.208581 -0.134151  0.158493
42  (2, 2, 2) -0.124501 -0.091669 -0.087500 -0.032443  0.439122  0.110770
43  (2, 2, 3) -0.112125  0.750724 -0.074750  0.242090 -0.099750 -0.130564
44  (2, 3, 0) -0.226980 -0.222886 -0.244419 -0.258608  0.412325 -0.247512
45  (2, 3, 1) -0.105581 -0.099750 -0.111387 -0.112125  0.106443  0.020007
46  (2, 3, 2) -0.081313 -0.099750  0.286692 -0.099750  0.701905  0.172537
47  (2, 3, 3)  0.283260 -0.001553  0.944312  0.261113  0.368475  0.264169
48  (3, 0, 0) -0.477070 -0.489049 -0.479968 -0.456889 -0.191802 -0.482806
49  (3, 0, 1) -0.354324  0.054954 -0.356650 -0.336801 -0.346720 -0.401431
50  (3, 0, 2) -0.173754  0.360836 -0.274550 -0.261672 -0.198257 -0.265132
51  (3, 0, 3) -0.302249 -0.282987 -0.296275 -0.207455 -0.296275 -0.284268
52  (3, 1, 0) -0.359574  0.105013 -0.344478 -0.347754 -0.362414 -0.326004
53  (3, 1, 1) -0.311254 -0.263596 -0.309338 -0.247376  0.291386 -0.279031
54  (3, 1, 2)  0.068462  0.462739 -0.029756  0.035910 -0.161441 -0.141675
55  (3, 1, 3) -0.229016  0.735801 -0.165811 -0.188276 -0.207324 -0.146437
```



```
56  (3, 2, 0) -0.247240  0.436511 -0.218051 -0.028955 -0.092885 -0.198505
57  (3, 2, 1) -0.163772 -0.137934 -0.198197 -0.124500  0.403712 -0.152158
58  (3, 2, 2)  0.207294 -0.050000  0.000000  0.187744  0.621448 -0.050000
59  (3, 2, 3)  0.291154  0.933014  0.297340  0.000000 -0.050000  0.253277
60  (3, 3, 0) -0.201617 -0.167690 -0.012305 -0.170433  0.424392 -0.198505
61  (3, 3, 1) -0.122622 -0.086579 -0.099750 -0.075000  0.778375 -0.099750
62  (3, 3, 2) -0.006499 -0.050000  0.000000  0.201429  0.930620 -0.050000
63  (3, 3, 3)  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
```

Evaluating Q-Table

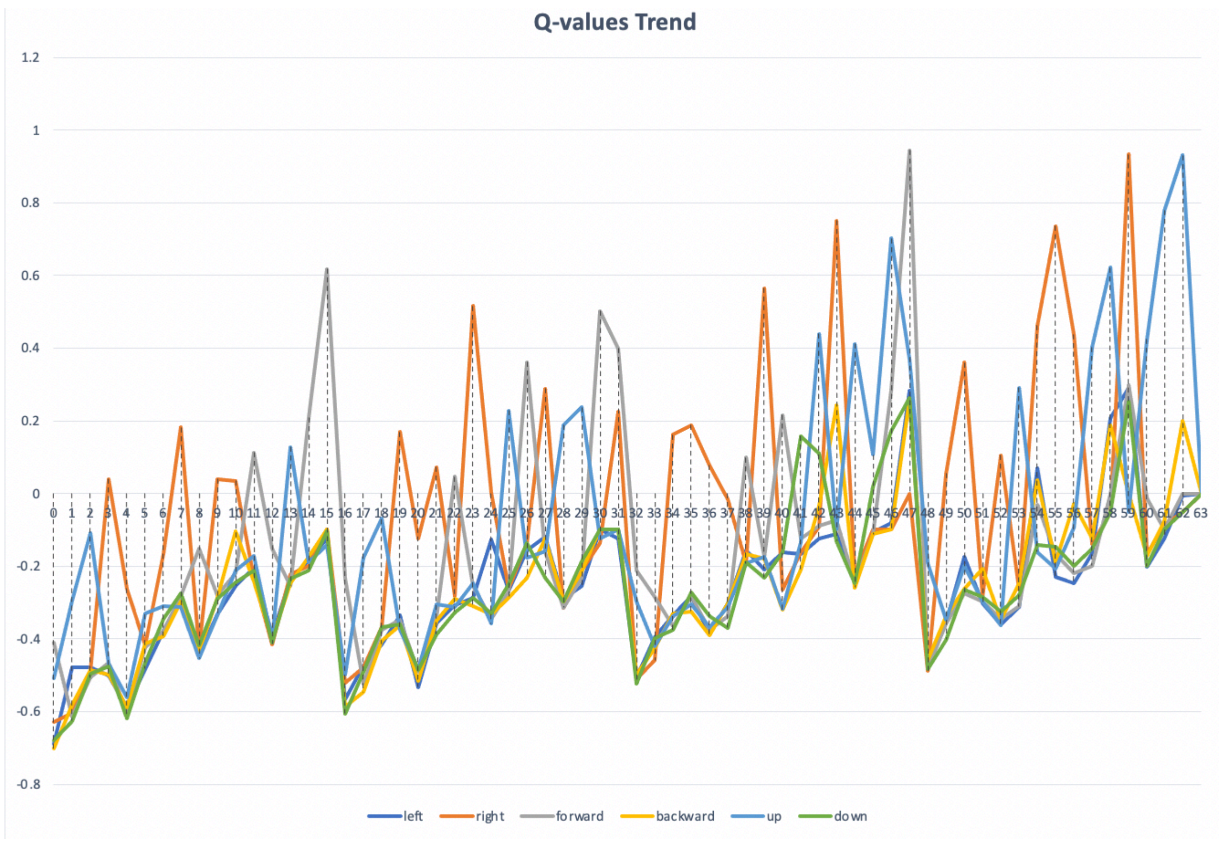
Using the above print output above, we can save a copy in csv and format it to analyse the overall trends in the action space q-table data. The formatted table is as follows:



	state	left	right	forward	backward	up	down
0	(0, 0, 0)	-0.689	-0.628	-0.411	-0.703	-0.509	-0.678
1	(0, 0, 1)	-0.478	-0.603	-0.613	-0.581	-0.294	-0.626
2	(0, 0, 2)	-0.479	-0.487	-0.506	-0.486	-0.109	-0.495
3	(0, 0, 3)	-0.500	0.041	-0.465	-0.498	-0.468	-0.477
4	(0, 1, 0)	-0.589	-0.261	-0.603	-0.589	-0.560	-0.618
5	(0, 1, 1)	-0.482	-0.419	-0.422	-0.412	-0.330	-0.465
6	(0, 1, 2)	-0.379	-0.167	-0.377	-0.392	-0.311	-0.344
7	(0, 1, 3)	-0.287	0.183	-0.283	-0.296	-0.311	-0.275
8	(0, 2, 0)	-0.442	-0.427	-0.152	-0.435	-0.453	-0.417
9	(0, 2, 1)	-0.332	0.040	-0.279	-0.288	-0.334	-0.290
10	(0, 2, 2)	-0.254	0.035	-0.220	-0.105	-0.213	-0.244
11	(0, 2, 3)	-0.204	-0.231	0.112	-0.241	-0.172	-0.211
12	(0, 3, 0)	-0.389	-0.414	-0.152	-0.399	-0.405	-0.411
13	(0, 3, 1)	-0.234	-0.223	-0.254	-0.248	0.128	-0.234
14	(0, 3, 2)	-0.183	-0.199	0.209	-0.174	-0.188	-0.212
15	(0, 3, 3)	-0.124	-0.100	0.619	-0.100	-0.143	-0.103
16	(1, 0, 0)	-0.567	-0.522	-0.235	-0.587	-0.498	-0.605
17	(1, 0, 1)	-0.472	-0.478	-0.524	-0.547	-0.176	-0.491
18	(1, 0, 2)	-0.414	-0.368	-0.376	-0.405	-0.069	-0.367
19	(1, 0, 3)	-0.334	0.169	-0.347	-0.362	-0.375	-0.360
20	(1, 1, 0)	-0.533	-0.123	-0.490	-0.517	-0.483	-0.485
21	(1, 1, 1)	-0.356	0.073	-0.341	-0.351	-0.304	-0.387
22	(1, 1, 2)	-0.308	-0.282	0.046	-0.290	-0.312	-0.328
23	(1, 1, 3)	-0.287	0.516	-0.254	-0.308	-0.248	-0.287
24	(1, 2, 0)	-0.127	0.001	-0.344	-0.333	-0.358	-0.331
25	(1, 2, 1)	-0.268	-0.262	-0.269	-0.288	0.229	-0.249
26	(1, 2, 2)	-0.155	-0.137	0.362	-0.231	-0.177	-0.137
27	(1, 2, 3)	-0.118	0.289	-0.162	-0.125	-0.159	-0.231
28	(1, 3, 0)	-0.290	-0.314	-0.314	-0.298	0.188	-0.296
29	(1, 3, 1)	-0.254	-0.211	-0.231	-0.214	0.237	-0.186
30	(1, 3, 2)	-0.100	-0.137	0.502	-0.100	-0.124	-0.100
31	(1, 3, 3)	-0.125	0.224	0.400	-0.100	-0.100	-0.100
32	(2, 0, 0)	-0.507	-0.510	-0.213	-0.498	-0.296	-0.523

33	(2, 0, 1)	-0.396	-0.459	-0.285	-0.425	-0.421	-0.398
34	(2, 0, 2)	-0.335	0.162	-0.365	-0.330	-0.337	-0.376
35	(2, 0, 3)	-0.282	0.189	-0.283	-0.326	-0.305	-0.271
36	(2, 1, 0)	-0.388	0.078	-0.374	-0.391	-0.368	-0.336
37	(2, 1, 1)	-0.302	-0.015	-0.338	-0.302	-0.313	-0.370
38	(2, 1, 2)	-0.156	-0.191	0.099	-0.168	-0.192	-0.189
39	(2, 1, 3)	-0.209	0.565	-0.174	-0.178	-0.174	-0.231
40	(2, 2, 0)	-0.160	-0.264	0.216	-0.320	-0.317	-0.161
41	(2, 2, 1)	-0.168	-0.161	-0.125	-0.209	-0.134	0.158
42	(2, 2, 2)	-0.125	-0.092	-0.088	-0.032	0.439	0.111
43	(2, 2, 3)	-0.112	0.751	-0.075	0.242	-0.100	-0.131
44	(2, 3, 0)	-0.227	-0.223	-0.244	-0.259	0.412	-0.248
45	(2, 3, 1)	-0.106	-0.100	-0.111	-0.112	0.106	0.020
46	(2, 3, 2)	-0.081	-0.100	0.287	-0.100	0.702	0.173
47	(2, 3, 3)	0.283	-0.002	0.944	0.261	0.368	0.264
48	(3, 0, 0)	-0.477	-0.489	-0.480	-0.457	-0.192	-0.483
49	(3, 0, 1)	-0.354	0.055	-0.357	-0.337	-0.347	-0.401
50	(3, 0, 2)	-0.174	0.361	-0.275	-0.262	-0.198	-0.265
51	(3, 0, 3)	-0.302	-0.283	-0.296	-0.207	-0.296	-0.284
52	(3, 1, 0)	-0.360	0.105	-0.344	-0.348	-0.362	-0.326
53	(3, 1, 1)	-0.311	-0.264	-0.309	-0.247	0.291	-0.279
54	(3, 1, 2)	0.068	0.463	-0.030	0.036	-0.161	-0.142
55	(3, 1, 3)	-0.229	0.736	-0.166	-0.188	-0.207	-0.146
56	(3, 2, 0)	-0.247	0.437	-0.218	-0.029	-0.093	-0.199
57	(3, 2, 1)	-0.164	-0.138	-0.198	-0.125	0.404	-0.152
58	(3, 2, 2)	0.207	-0.050	0.000	0.188	0.621	-0.050
59	(3, 2, 3)	0.291	0.933	0.297	0.000	-0.050	0.253
60	(3, 3, 0)	-0.202	-0.168	-0.012	-0.170	0.424	-0.199
61	(3, 3, 1)	-0.123	-0.087	-0.100	-0.075	0.778	-0.100
62	(3, 3, 2)	-0.006	-0.050	0.000	0.201	0.931	-0.050
63	(3, 3, 3)	0.000	0.000	0.000	0.000	0.000	0.000

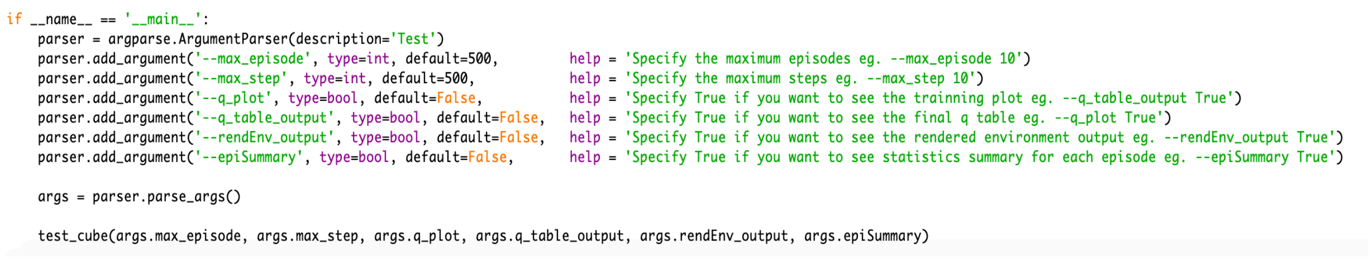
From what we see in the table above, we can hypnotise that the q-values increase as episodes increase since the overall trend seems to be going from orange (signifying lower values overall) to green (signifying higher values overall).



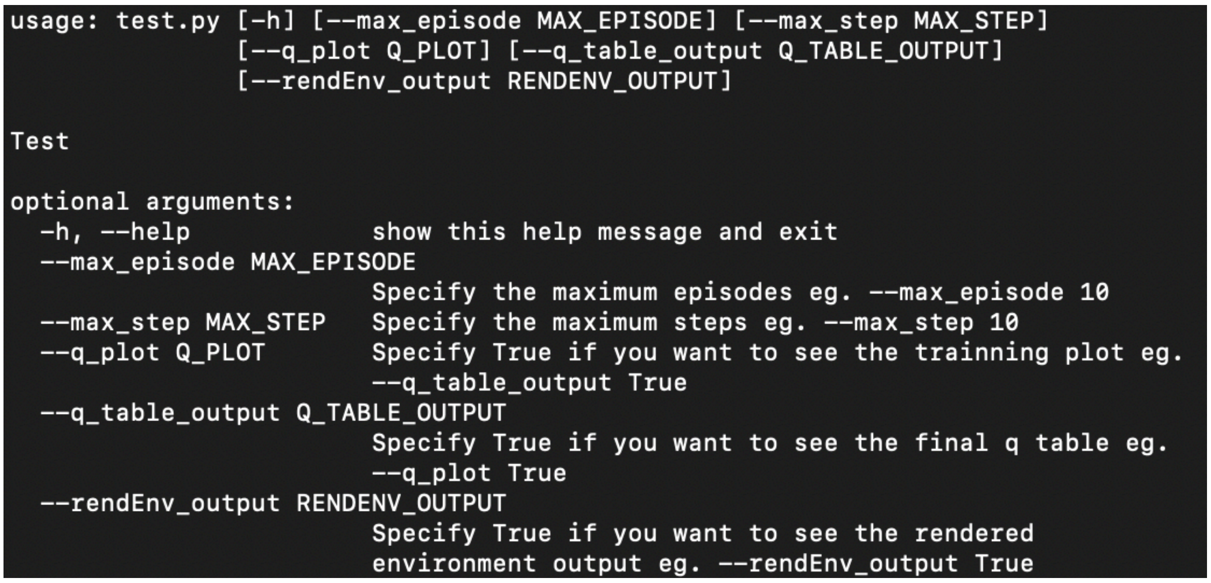
Using the values from our q-table, we can plot the above graph to understand the trends in the q-values of the actions respectively. As we can see the general trend agrees with our hypothesis from the previous table: the values increase as the state increases.

User Inputs

The code is structured such that the user can customise their desired output format.


```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test')
    parser.add_argument('--max_episode', type=int, default=500,        help = 'Specify the maximum episodes eg. --max_episode 10')
    parser.add_argument('--max_step', type=int, default=500,           help = 'Specify the maximum steps eg. --max_step 10')
    parser.add_argument('--q_plot', type=bool, default=False,          help = 'Specify True if you want to see the trainning plot eg. --q_table_output True')
    parser.add_argument('--q_table_output', type=bool, default=False,  help = 'Specify True if you want to see the final q table eg. --q_plot True')
    parser.add_argument('--rendEnv_output', type=bool, default=False,  help = 'Specify True if you want to see the rendered environment output eg. --rendEnv_output True')
    parser.add_argument('--epiSummary', type=bool, default=False,      help = 'Specify True if you want to see statistics summary for each episode eg. --epiSummary True')

    args = parser.parse_args()

    test_cube(args.max_episode, args.max_step, args.q_plot, args.q_table_output, args.rendEnv_output, args.epiSummary)
```

All the arguments and their set help messages are shown below:


```
usage: test.py [-h] [--max_episode MAX_EPISODE] [--max_step MAX_STEP]
               [--q_plot Q_PLOT] [--q_table_output Q_TABLE_OUTPUT]
               [--rendEnv_output RENDENV_OUTPUT]

Test

optional arguments:
  -h, --help            show this help message and exit
  --max_episode MAX_EPISODE
                        Specify the maximum episodes eg. --max_episode 10
  --max_step MAX_STEP   Specify the maximum steps eg. --max_step 10
  --q_plot Q_PLOT       Specify True if you want to see the trainning plot eg.
                        --q_table_output True
  --q_table_output Q_TABLE_OUTPUT
                        Specify True if you want to see the final q table eg.
                        --q_plot True
  --rendEnv_output RENDENV_OUTPUT
                        Specify True if you want to see the rendered
                        environment output eg. --rendEnv_output True
```

To generate only the summary of average steps and rewards per episode, with the default 500 max episode and max step, run the following:

```
python test.py
```

To generate the summary of episodes, steps, and rewards, with the default 500 max episode and max step, run the following:

```
python test.py –epiSummary True
```

To generate output with specified max episode and max step, run the following:

```
python test.py --max_episode 500 --max_step 500
```

To generate a plot of the learning progress, run the following:

```
python test.py --q_plot True
```

To generate the final q-table, run the following:

```
python test.py --q_table_output True
```

To generate an output including the rendered environment, run the following:

```
python test.py --rendEnv_output True
```

The arguments above can be joint together as per the user’s choice of output.