BITS Pilani
K K Birla Goa Campus

Name: Sarthak Dalmia

ID: 2018B3A70290G

Artificial Intelligence (CS F407)

BITS, Pilani, Goa Campus

Assignment No. 2

December 2, 2021

Training Q-Learning using Monte Carlo Tree Search to play Connect-4

Table of Contents

1. Abstract

This report focuses on implementing both, the Monte Carlo Tree Search and Q-learning algorithms to play the game of Connect 4. We run MCTS algorithm for 40 and 200 simulations and show that MC_{200} wins most of the time against MC_{40}. Lastly, we train Q-learning algorithm against our MC_n algorithm, for n ranging from 0 to 25, for a simplified Connect 4 game of size 5 columns x r rows using afterstates.

2. Overview

The report is organized as follows. Section 1 contains the problem specification. Section 2 contains the detailed implementation of the Monte Carlo algorithm for a game of 5 columns x 6 rows for 200 and 40 simulations. Section 3 contains the implementation of Q-learning in way that expedites its convergence against MC_n for n between 0 and 25. Lastly, Section 4 contains the modified Q-Learning algorithm to win against MC_n for a range of n.

2.1 Problem Specification

In this assignment, Monte Carlo tree search and Q-Learning is to be implemented to play the game of Connect 4. For Monte Carlo, the board size is 5 columns x 6 rows and number of simulations are 40 and 200. For Q-Learning, Monte Carlo must be used to train the algorithm and should win against the later.

3. Monte Carlo Tree Search

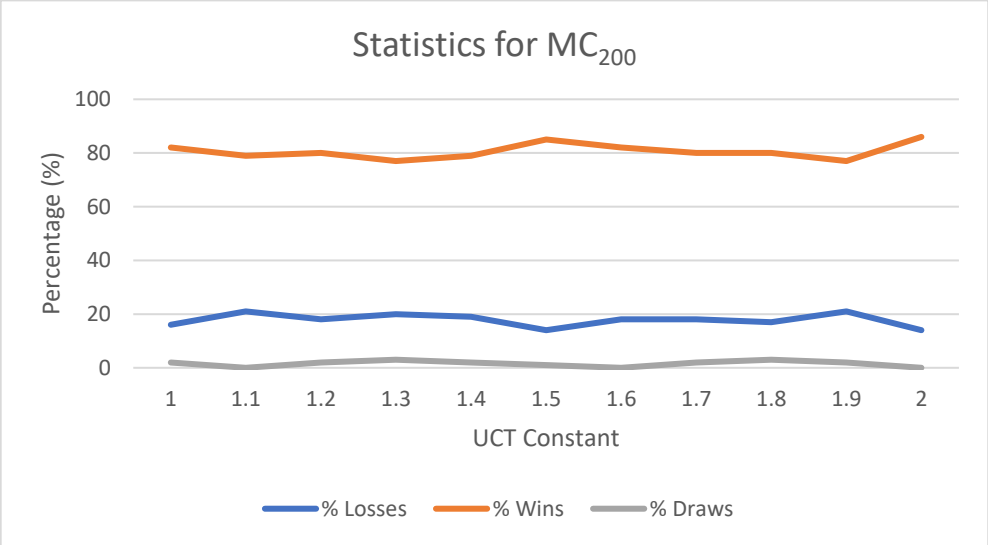
3.1 Implementation

A tree of depth 4 is created via simulations at each time step including the first one. This entails a possibility of having a child-node already existing in the tree (with some value) before we begin its simulations. As usual we return the move to take, as the move with maximum number of playouts. Each node's value is calculated via UCT function with rewards rather than wins as a parameter. If multiple nodes have the same value, then we

choose a move (or node) randomly. In terms of expansion policy, we can either choose to create all the immediate children or only one child. Here, all the children are created. This is simply because it makes it easy to code as we now don't have to check each time whether we have all the children from a particular node and or not and if not do we select from the existing child node or expand via some other policy.

3.2 Parameters and Evaluation

The algorithm with 200 simulations benefits first and foremost with the **number of simulations.** More number of simulations helps in converging into the best action at each step more accurately. With 40 simulations, for an epsilon, there is a chance that the exploration part of the algorithm heavily effects our final choice, which may not then be the best choice. The **terminal rewards** granted were changed to make the game finish in fewer moves and to increase win percentage of the MC_{200} algorithm. These were changed for both the 40 simulations and 200 simulations algorithm and calculated over a range of **UCT constant** values for 100 iterations each.



The above graph shows the average win loss and draw percentage over different reward maps. It was found that with changing of the UCT constant value there is no drastic change in percentage value and thus the default of 1.4 is taken for further runs of the algorithm. After choosing the UCT value as 1.4 we tabulate the win, draw, loss percentage and number of moves for a few of the reward maps tried.

Rewards		Statistics for MC_{200}			
Loss	**Draw**	**Average No. of Moves**	**% Losses**	**% Draws**	**% Wins**
-1	**0**	22.03	14.75	13.04	72.21
-0.5	**-0.5**	21.21	19.21	0.77	80.02
0	**0**	19.58	19.21	0.84	79.95
-1	**-1**	20.71	17.76	1.53	80.71
-1.5	**-1.5**	20.63	19.38	1.21	79.41

It was found that the reward structure of -1, 0, 1 for Loss, Draw and Win respectively results in a minimum percentage of loss for the 200 simulations algorithm. Moreover, for the reward structure -1, -1, 1 for loss, draw and win respectively it seen that the win percentage is maximum and the number of moves required are fairly close to the minimum found. Also, in the later case the draw percentage was considerably lower than that of former case.

3.3 Final MCTS Parameters

Finally, the reward map of **-1, -1, 1 for Draw, Loss and Win** respectively is chosen. An **UCT constant value of 1.4** is chosen.

4. Q-Learning

4.1 Implementation

Both the vanilla Q-Learning with training state-action pairs is trained and an Afterstates algorithm of Q -Learning is trained. These two algorithms are compared on the bases of there win percentage against MCTS algorithm. For both the algorithms epsilon greedy policy is used for next state selection. That is after finding the action for maximum state-action pair value for vanilla Q-Learning and maximum afterstates value for afterstates Q-Learning, we choose that action with an epsilon greedy policy.

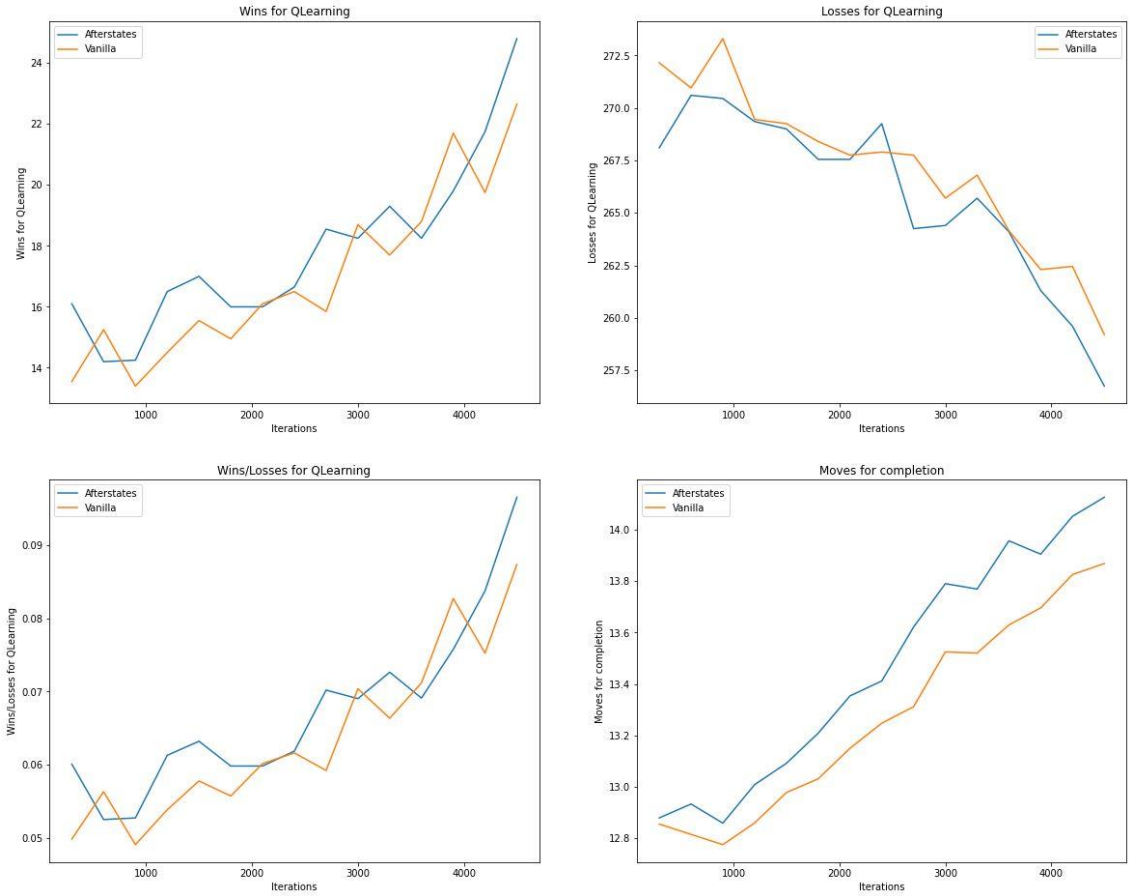
4.1.1 Vanilla Q-Learning Value Updation

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * (R_{t+2} + \gamma * max_a Q(S_{t+2}, a) - Q(S_t, A_t))$$

4.1.2 Afterstates Q-Learning Value Updation

$$V(S_t) = V(S_t) + \alpha * (R_{t+1} + \gamma * max_s V(S_{t+1}) - V(S_t))$$

To choose between afterstates and vanilla Q-Learning algorithm, the following graphs are plotted.



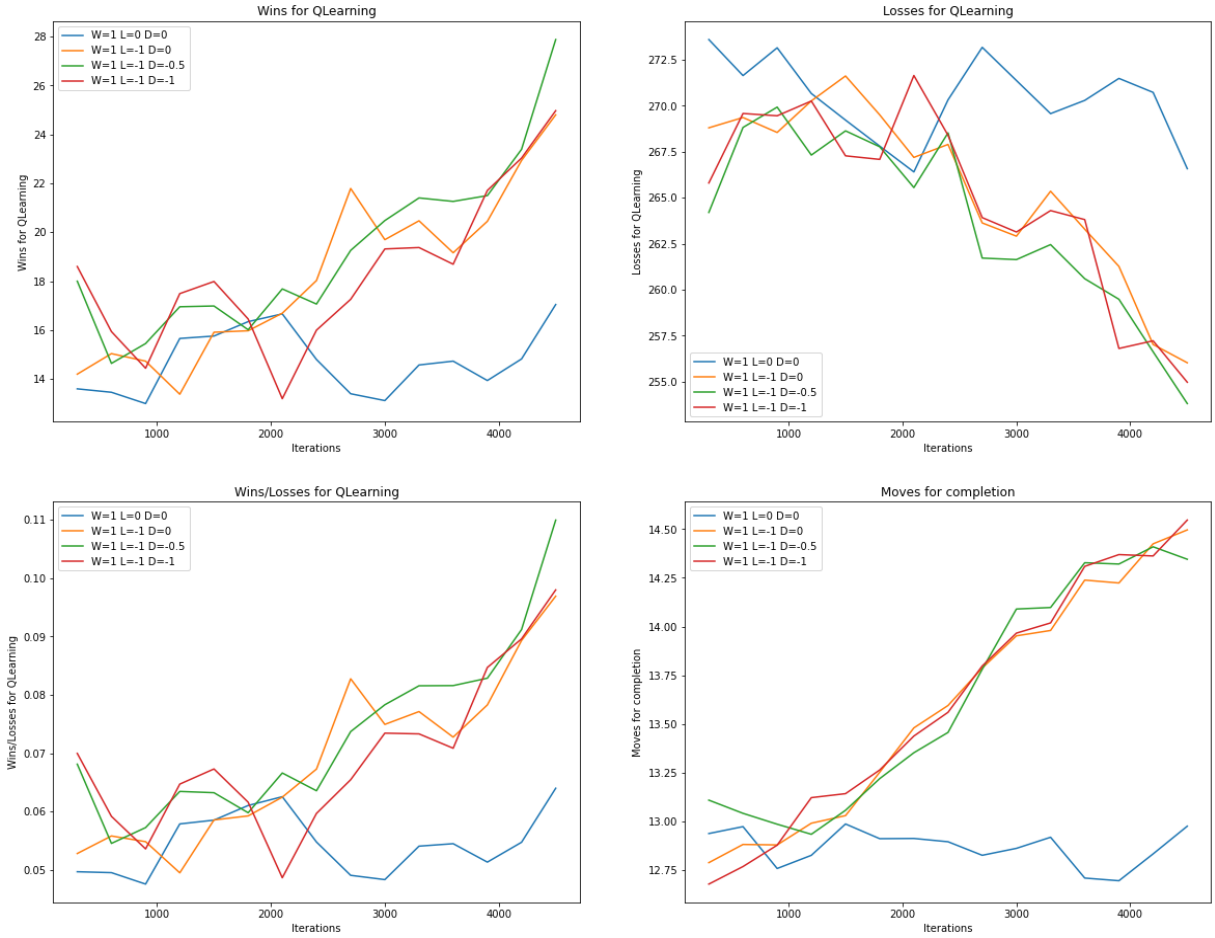
The above graphs were plotted with all else held constant environment. That is the values for alpha, gamma, epsilon and the reward map was same for both the algorithms during computation. As can be seen from the graph, afterstates performs better than vanilla in terms of win/loss ratio against the MC algorithm. In the number of moves metric, vanilla performs better. The latter metric should be taken with a pinch of salt though. According to the algorithm, number of moves are recorded for every game, no matter if Q-Learning wins and losses. The graph stating the absolute values for wins and losses shows us that Q-Learning is losing more games than it is winning by a big margin. This, adding to the number of moves graph, gives us a new inference that the afterstates algorithm might be making it farther in the games in which it has lost as compared to the vanilla algorithm. This thus can be interpreted as the fact that vanilla algorithm is losing faster as compared to the afterstates algorithm. Thus, finally, the afterstates algorithm is chosen. In the next section we chose optimal parameters for this algorithm.

4.2 Parameters and Evaluation

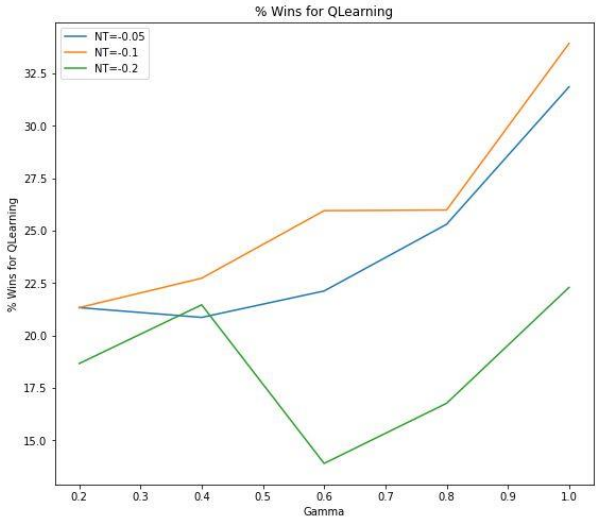
Afterstates were useful because firstly they save memory, it saves training time because it condenses multiple state-action pairs which lead to same single value which will themselves be updated several times. Also, from the graphs shown below we see that wins of afterstates algorithm is higher than that of vanilla algorithm. Next, different **reward maps**, **gamma(γ)** and **epsilon(ε)** values are tried. **Learning rate (α)** was fixed at 0.1 for each of these trials. Parameters chosen for the MCTS are that from the previous part, that is, reward map of -1, -1, 1 for loss, draw, and win and UCT constant of 1.4. The non-terminal reward for MCTS algorithm is kept at 0.

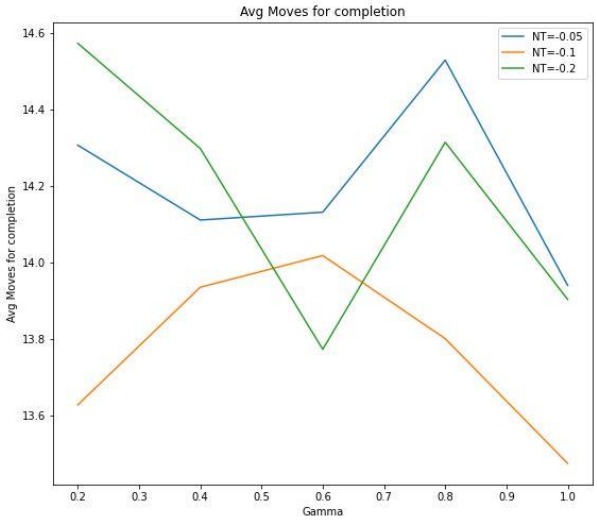
Now we want to select the best reward map for our algorithm. For this we keep the rest of our parameters for Q-Learning that is gamma, epsilon and alpha as same for each reward map. To generalize our results, average graphs are created. That is, alpha is fixed at 0.1 as mentioned before. Epsilon is progressively decreased and for each run of a set of iterations. Next, five different gamma values are considered between 0.2 and 1, both inclusive, and then averaged over them. Four different reward maps are tested, listed as {Draw, Loss, Win}: {0, 0, 1}, {0, -1, 1}, {-0.5, -1, 1} and {-1, -1, 1}. After this the best

gamma is selected by further analysis.



The **reward map for draw, loss and win** chosen is **-0.5, -1 and 1** which is, as can be seen from the above graphs, is the best in terms of wins in Q-Learning. From this we take out the best possible gamma and reward map for Q-learning based on the number of moves and win percentage of the Q-Learning. These graphs were computed with decreasing epsilon and number of iterations at each epsilon at 4500. This is performed for 3 non-terminal reward values -0.05, -0.1 and -0.2. This is done so that the Q-Learning wins in the minimum number of moves. Finally, average is calculated for each of these games and plotted against the varying gammas.



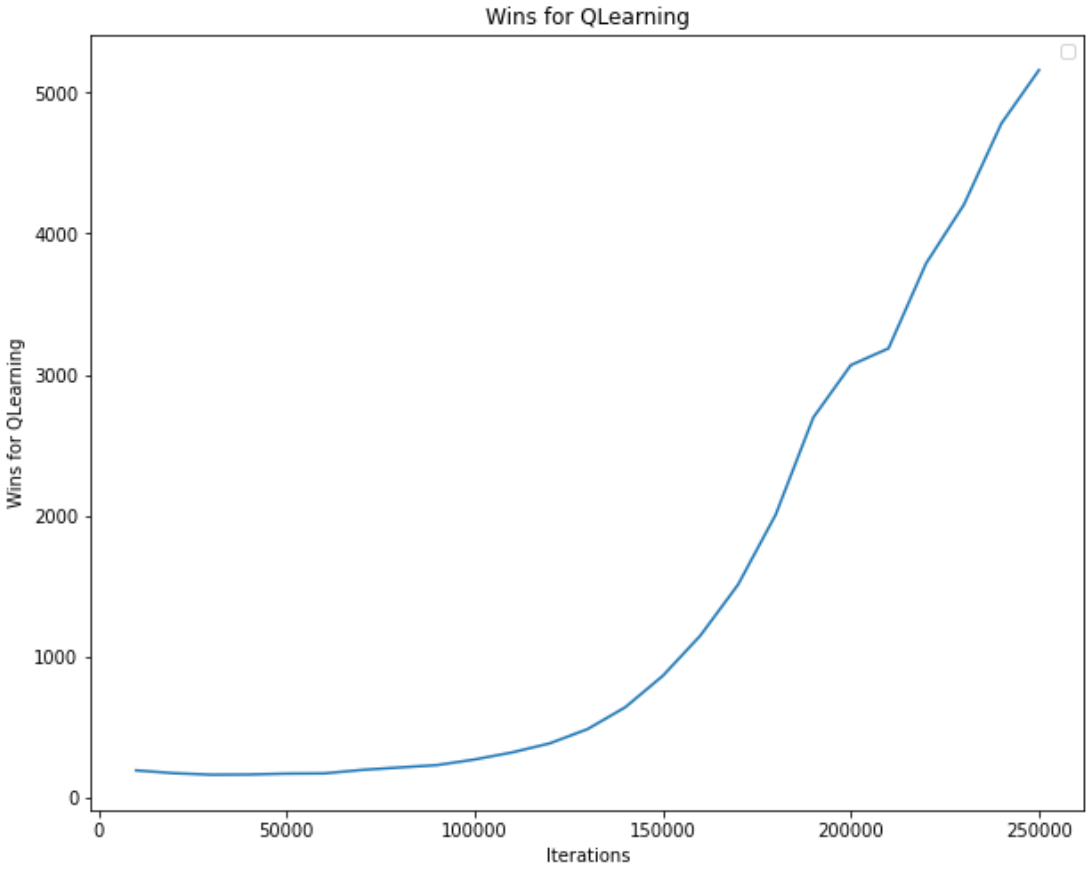


As can be seen in the above graph, based on the reward for non-terminal states that is taken, it can be seen that the best gamma and best non-terminal award for the Q-Learning is **gamma (γ) = 1** and **non-Terminal reward = -0.1**. One liberty taken to reduce the number of iterations is that we train our Q-learning against MC_{20} instead of MC_{25}.

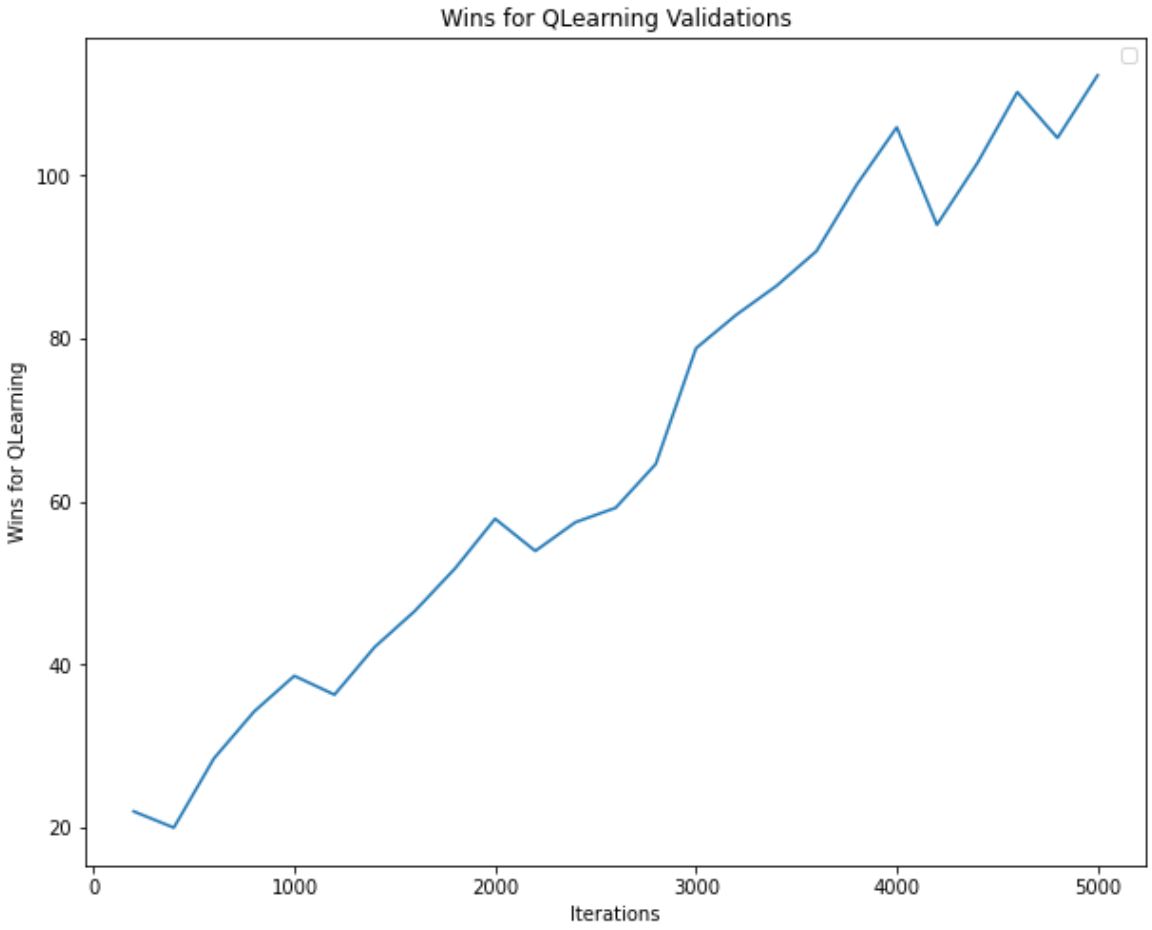
4.3 Final Q-Learning Algorithm

For choosing the best possible value for gamma and epsilon, the number of wins Q-learning secures against Monte Carlo tree search for 0 to 25 simulations was considered. Adding to the previous criterion, training time and number of moves to win is also considered. Q-Learning is pitted against MC for 0 to 25 with an interval of 5. It is found out that the less the number of simulations for the MCTS, the higher is the win percentage of Q-Learning is as can be seen in the previous graphs, which was as expected. Moreover, running the algorithms for a greater number of iterations increases the win percentage suggesting that convergence was not yet achieved. Now for our value map we have **decreasing epsilon, gamma as 1, reward map as -1 for loss, -0.5 for Draw and 1 for Win. Non-Terminal** state has a reward of **-0.1. Alpha** is fixed at **0.1.** Final value map is stored in a datafile and is calculated by running the above-mentioned Q-Learning algorithm for 10000 iterations. After this run, epsilon of Q-Learning was set to 0, and to accurately test the performance of Q-Learning against MCTS algorithm, it is validated by making it play against the MC agent again for 200 iterations.

The below graphs show the training graph of the Q-Learning algorithm. The total number of iterations comes out to about 250000 and took about 12 hours to execute. This graph has been smoothed using an ema function.



Following is the validation graph for the above trainer. It was run for about for 200 iterations in each step which comes out to about 5000 iterations in total.



5. Reducing the datafile size.

Each state of the game (or board) is represented such that first player is “1” and second player is a “2” in the grid. Now we concatenate all the elements in each row to make one string, which is an integer. This is then saved to a file using the pickle function which stores objects directly as their binary representation rather than converting them into strings. This is saves memory and reduces file size.