CSIS, Bits Pilani K.K. Birla Goa Campus

Artificial Intelligence (CS F407)

Programming Assignment 2 (Report)

Name – Sharanya Ranka

Roll No. – 2018A7PS0215G

Submission Date – 4th December, 2021.

Contents

List of Figures

1 Question 1 - MC_{200} vs MC_{40}

1.1 Discussion

The UCT (Upper Confidence Bound applied to Trees) uses the following formula to calculate an estimate of the upper bound of the "value" of a state[1]:

$$UCT(s) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{1}$$

Here we can consider the $\frac{w_i}{n_i}$ term as the estimate's *mean* value and $\sqrt{\frac{\ln N_i}{n_i}}$ as the *uncertainty* in the mean.

If we increase the value of c, this increases the upper-bound estimates, and consequently, the MCTS agents will search more *evenly* among child states.

Some of the parameters that can be tweaked here are:

- ***uct c value*** - This value controls the upper-bound estimates of the"goodness of" states. A higher c value would encourage exploration more *evenly* among child states. This will help MC_n agents with larger n to get better estimates of the mean value of the child states, eventually helping it to select better actions. We perform experiments to get the best c value
- ***final action selection strategy*** - This strategy controls how the actual action is selected, given the uct estimates of the child states. 2 strategies are tested:
  - *ROBUST CHILD selection* - Here the child node visited the most number of times is selected. This doesn't depend directly on the uct score of the child (although it depends indirectly, as children with higher uct-scores are explored more)
  - *BEST CHILD selection* - In this strategy, we use only the $mean$ estimate to get a sense of which action is the best (as exploration is done). Here, we select based on the best value of $UCT(s)$ with $c = 0$.

1.2 Experiments

As mentioned before, experiments were conducted on 2 parameters. For each *final action selection strategy*, we test *uct c* values from 0 to 1.5 in increments of 0.1. We have a total of ($2 \times 16 = 32$ runs).

For each run, 100 games were played. For the first 50 games, MC_{200} was player 1 and for the last 50 games MC_{200} was player 2.

[1] It is important to note that this formula *does* differentiate between wins and draws, as wins get +1 while draws get +0.5 points. If one action leads the agent to victory while the other leads to a draw, the *mean value* term will be higher for the victory-action, and consequently the agent will choose that action.

Time taken for runs - For this we assume the average number of moves per game = 14 (7 moves per player). This is about half of the maximum possible num of moves (=30). For each run, therefore we get ($100 \times 7 \times (240 + 50)ms \approx 200\ secs$) (refer to time plots to get (240+50)ms) per run and ($200 \times 32 \approx 1hr\ 40min$) for all the runs.

Once this is done, and data is saved, we plot the number of wins, draws and losses of MC_{200} vs MC_{40} (summing over outcomes as player 1 and player 2). This is shown as a bar chart below.

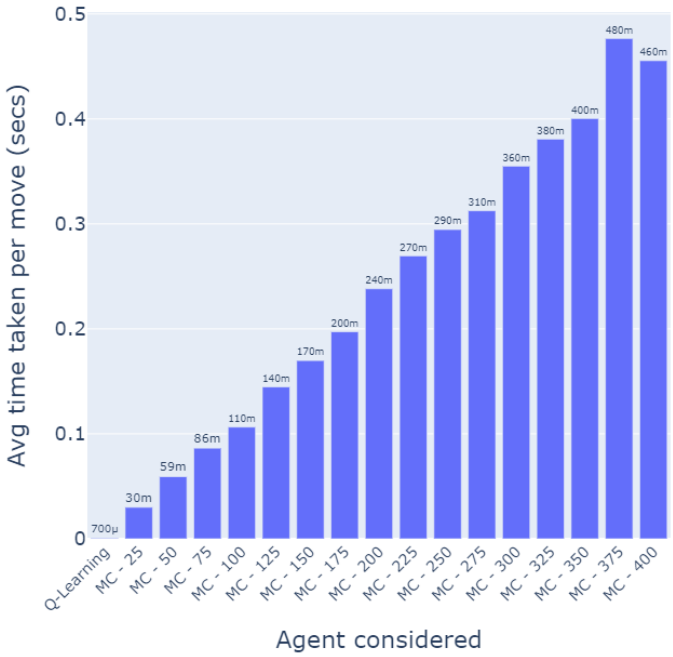
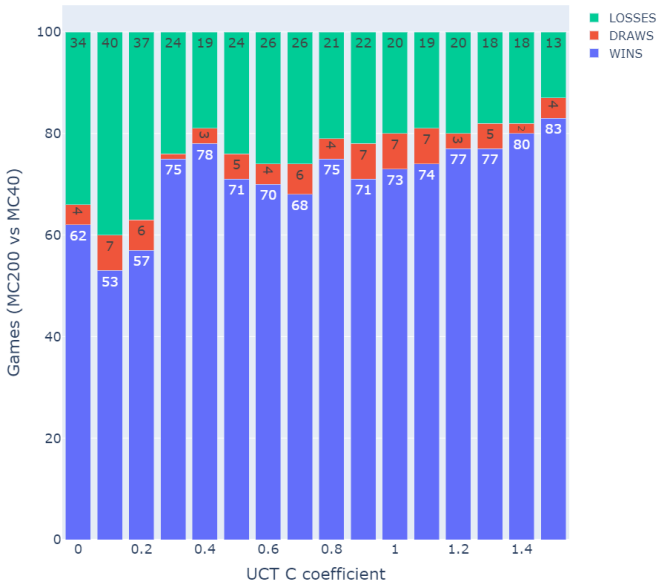
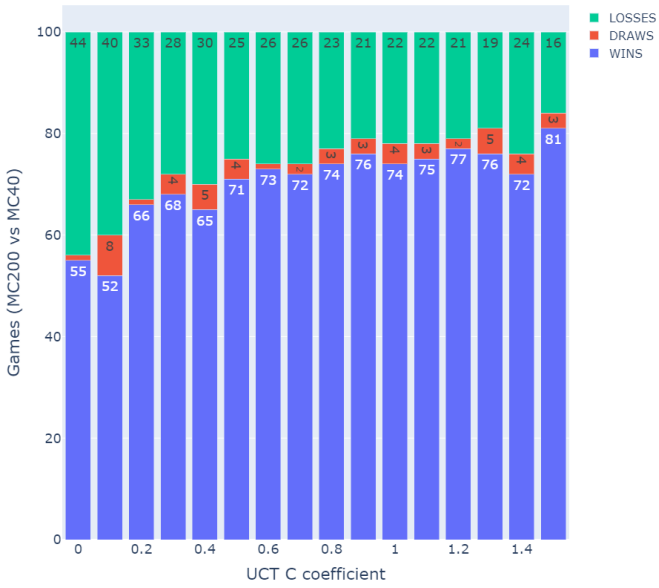


Figure 1: Graph of time taken (secs) by various agents to make a move

Figure 2: Here we show the results of games played by MC_{200} against MC_{40} (from MC_{200}'s perspective)



(a) *ROBUST CHILD selection* - Here we see that as the uct c coefficient increases, the number of games won against MC_{40} increases as well.



(b) *BEST CHILD selection* - Here we see that as the uct c coefficient increases, the number of games won against MC_{40} increases as well

1.3 Results

We see that the graphs for both *ROBUST CHILD selection* and *BEST CHILD selection* strategies are very similar. A higher $uct\ c$ value helps MC_{200} beat MC_{40} as the children states are explored more evenly, and better estimates are formed for the child nodes. After seeing the results, we select $uct\ c = 1.5$ for our MC_n agents.

To select between *ROBUST CHILD selection* and *BEST CHILD selection*, we note that at $uct\ c = 1.5$, *ROBUST CHILD selection* has more wins, and thus we will select this strategy for next experiments.

2 Question 2 - Q-Learning training and convergence

2.1 After modification ($r = 2, 3,\ n = 0 - 25$)

2.1.1 Discussion

The experiments have been performed and graphs shown for $r = 2, 3$. We show that, while training against MC_{25} there are signs that the agent converges for $r = 2$, but not for $r = 3$. We have trained the Q-Learning agent only against MC_{25} as this is the strongest agent in the given constraints (on n). Similar graphs can also be generated against other agents, but I don't think it will yield much new information.

Some of the parameters that can be tweaked are :

- α - The parameter that decides the exponential decay of previous experience. We perform experiments to check which α value is the best.
- ϵ - The parameter that decides the probability with which random actions should be taken (for exploration). We perform experiments to check which ϵ value is the best.
- γ - Discounting factor - how much should the estimated future rewards be weighted. As the games are finite (i.e. proper episodes can be defined), a natural $\gamma = 1$ is used. We do not experiment on this parameter.
- $behaviour\ policy$ - How the actions are selected. Currently an ϵ-greedy policy is used with constant ϵ throughout the episodes. We do not experiment with this parameter.
- $reward\ structure$ - What rewards are given to the QLearning agent. Currently, keeping in mind that the QLearning agent is supposed to win with minimum moves, the following reward structure is used:
  - *WIN* : +100
  - *LOST* : -100
  - *DRAW* : -10
  - *GAME-IN-PROGRESS* : -1

A reward of -1 given for "GAME-IN-PROGRESS" helps the QLearner choose actions that will help it win quicker. This is equivalent of giving the mouse a reward of -1 every move if it is still in the maze.

A reward of -10 is given for a draw, as we want to avoid draws where possible, but still prefer them to losing.

We do not experiment on the reward structure.

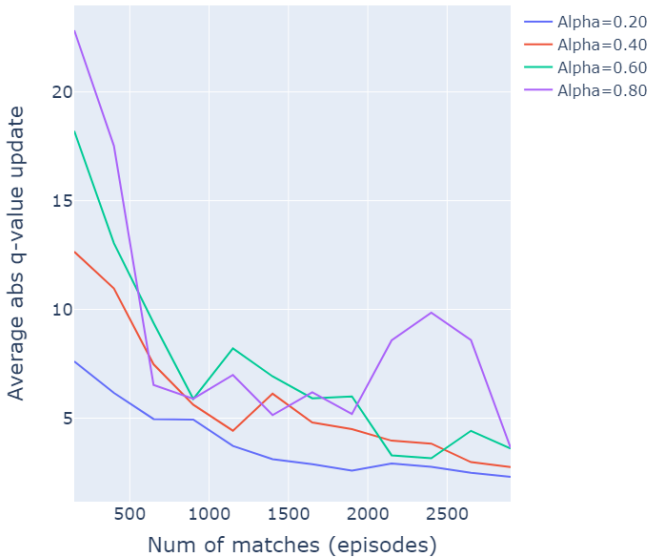
2.1.2 Experiments

We experiment on the α and ϵ parameters to see which parameters help agent train quicker.

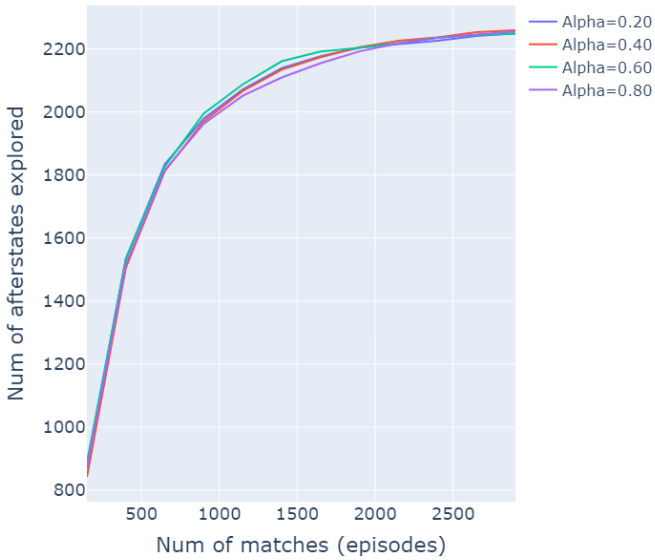
For $r = 2$, given that the number of unique afterstates encountered are small (≈ 2200), we try values for α - [0.2, 0.4, 0.6, 0.8] and values for ϵ - [0.0, 0.2, 0.4, 0.6, 0.8]. These values are tested separately and not in combination (to reduce time taken). We have (4 + 5) values to test, and simulate 3000 episodes for each. Each test took $3k \times 5 \times 30ms = 450\ secs$ approximately.

For $r = 3$, the number of unique afterstates encountered are more ($\approx 40k$), we try values for α - [0.2, 0.4] and values for ϵ - [0.0, 0.1, 0.2]. This can be justified since we see from the previous experiment that large α causes large updates late in the training, and agents with large ϵ don't fare well against MC_n. These values are tested separately and not in combination (to reduce time taken). We have (2 + 3) values to test, and simulate 10k episodes for each. Each test took $10k \times 7 \times 30ms = 2100\ secs$ approximately.

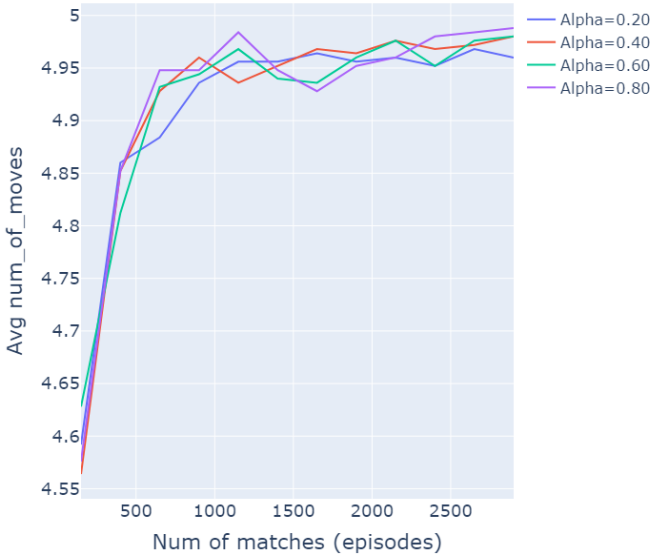
Figure 4: Results of experiments testing various values of α for the 2X5 board. Trained against MC_{25}, here we can see convergence from the absolute q-value update and number of unique afterstates explored. The number of losses against the MC_{25} agent also drops drastically with episodes



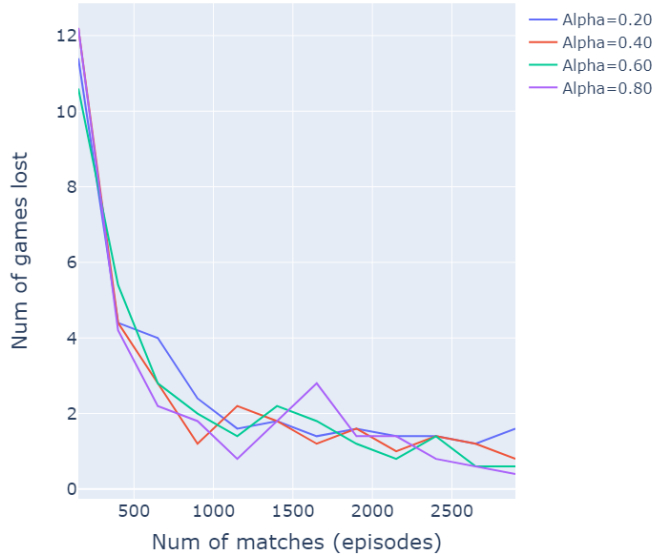
(a) Here we can see that $\alpha = 0.2$ shows the least absolute q-value update by the end of 3000 episodes



(b) Here we can see that all α values show similar rate of exploration of new afterstates. Also, the number of unique afterstates as second player is close to 2200

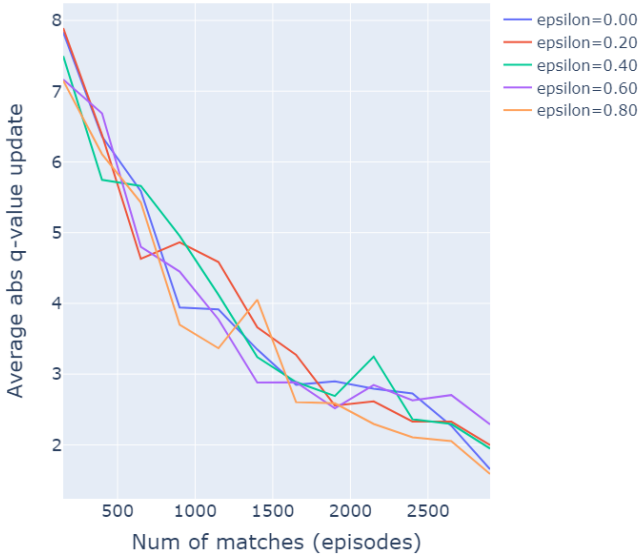


(c) Even with the reward structure mentioned in the report, we see the number of moves going up with number of episodes. As most games result in a draw, the avg number of moves is very close to 5.

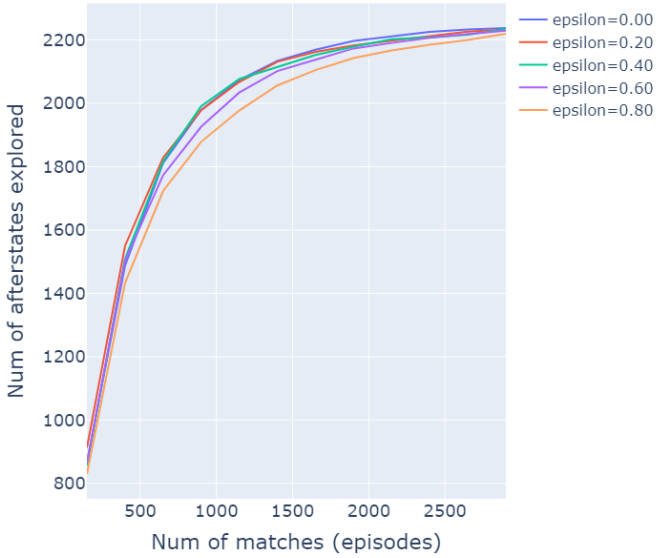


(d) The number of losses (per 50 games played with MC_{25}). Along with the previous graph, we find that the reason why number of moves increases, is because the agent is learning not to lose easy games. The α values give us no special information here.

Figure 5: Results of experiments testing various values of ϵ for the 2X5 board. Trained against MC_{25}. The number of losses against the MC_{25} agent drops drastically with episodes for $\epsilon = 0$

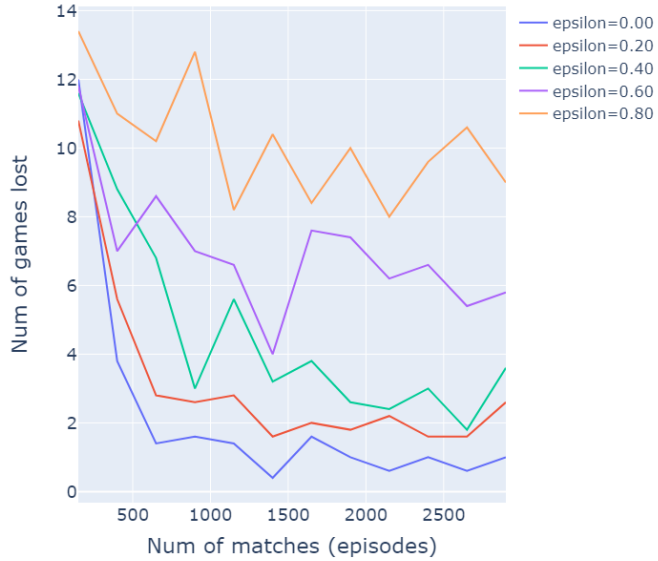


(a) Here we see that for all values of ϵ, the average absolute q-value update decreases



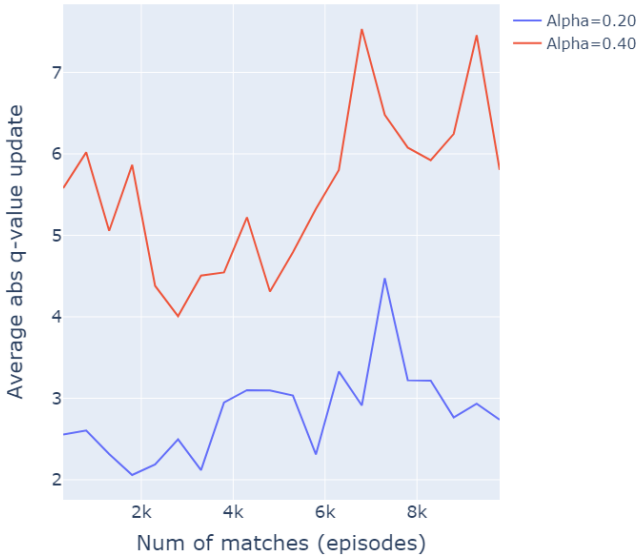
(b) Again, just like in the experiments with α, all ϵ values explore afterstates at a similar rate. The number of unique afterstates as second player is close to 2200

(c) The same comment mentioned about reward structure vs avoiding losses applies here. Interestingly, $\epsilon = 0$ seems to perform the best. As most games result in a draw, the avg number of moves is very close to 5.

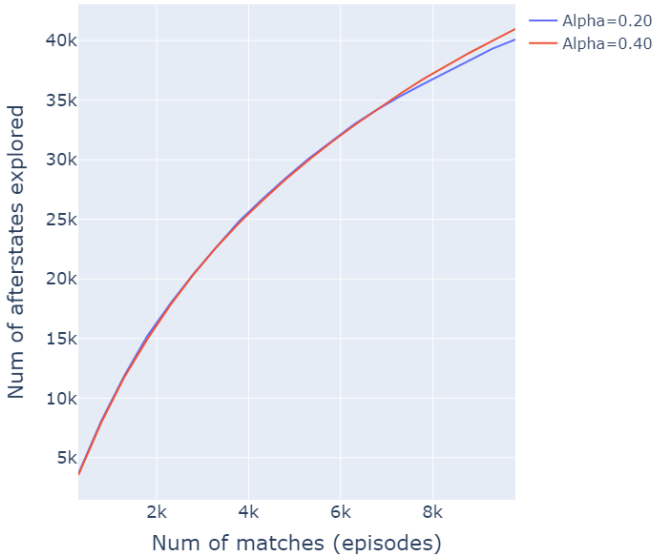


(d) Number of losses (per 50 games played with MC_{25}). The previous graph, along with this one suggest that $\epsilon = 0$ is the best choice for the Q-Learning agent. This may be because the reward structure is highly negative (only wins are +100). This means the negative q-value estimates will drive exploration even with $\epsilon = 0$

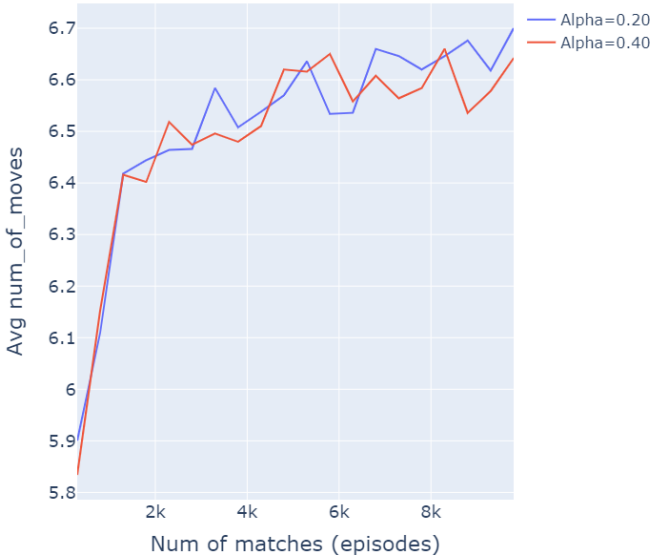
Figure 6: Results of experiments testing various values of α for the 3X5 board. Trained against MC_{25}. Here we see that there's no convergence. The afterstates are yet being actively explored, even after episode 10k. The average absolute q-value update doesn't decrease as expected for convergence. On the other hand, the number of games lost decreases significantly (by $\approx$ 20 games per 100 games), suggesting that the agent is learning, but hasn't converged yet



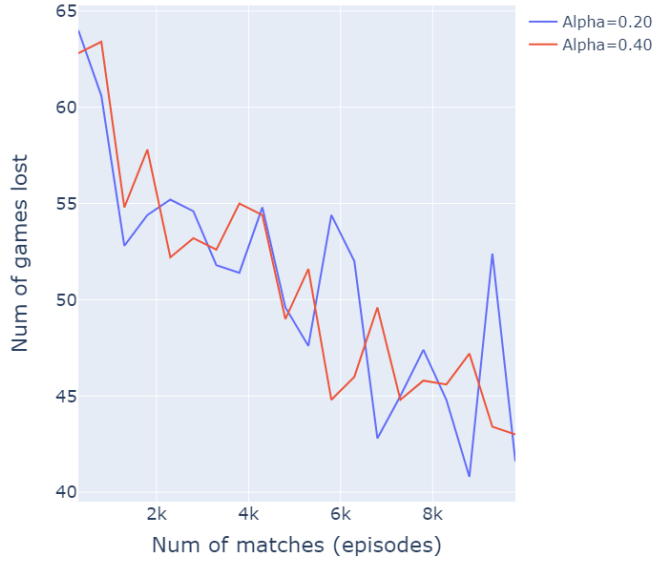
(a) Here we can see that $\alpha = 0.2$ shows the least absolute q-value update by the end of 10k episodes. Even in the beginning, the updates for $\alpha = 0.2$ are smaller, but this is expected as α controls the update value for q-value



(b) Here we see that for both $\alpha = 0.2$ or 0.4, the rate of exploration of afterstates is approximately same. $\alpha = 0.4$ does have a slight edge towards the end ($\approx$ 1k more afterstates). By the end of episode 10k, around 40k afterstates are explored as player 2

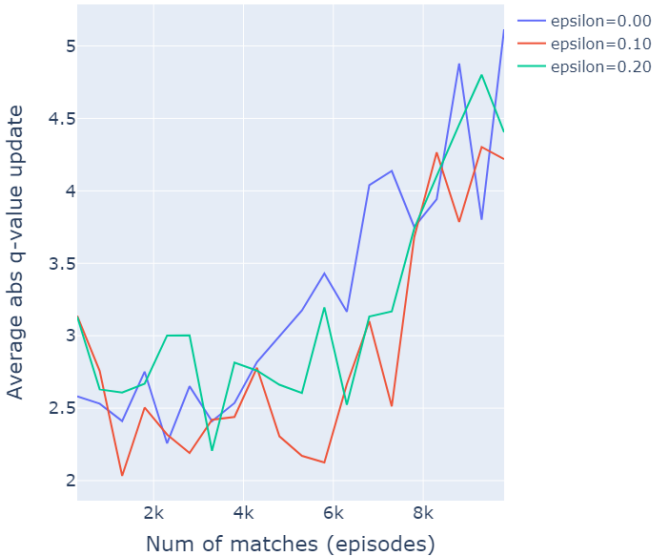


(c) Here also we see the number of moves made by the Q-Learning agent increase with episodes. As many games result in a draw, the avg number of moves is very close to 7.

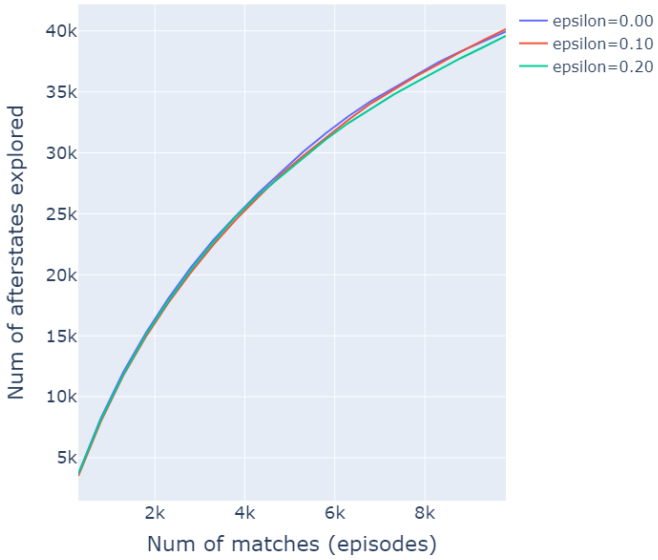


(d) The number of losses (per 100 games played with MC_{25}). The α values give us no special information here.

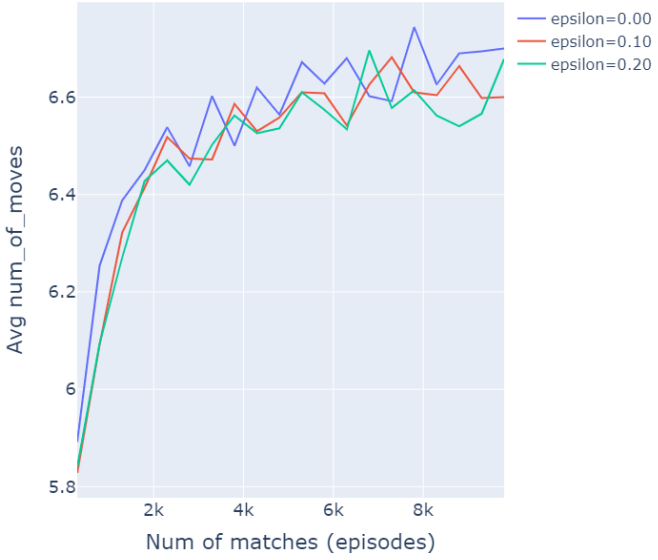
Figure 7: Results of experiments testing various values of ϵ for the 3X5 board. Trained against MC_{25}. Convergence is not seen. The number of losses against the MC_{25} agent drops drastically with episodes for $\epsilon = 0$



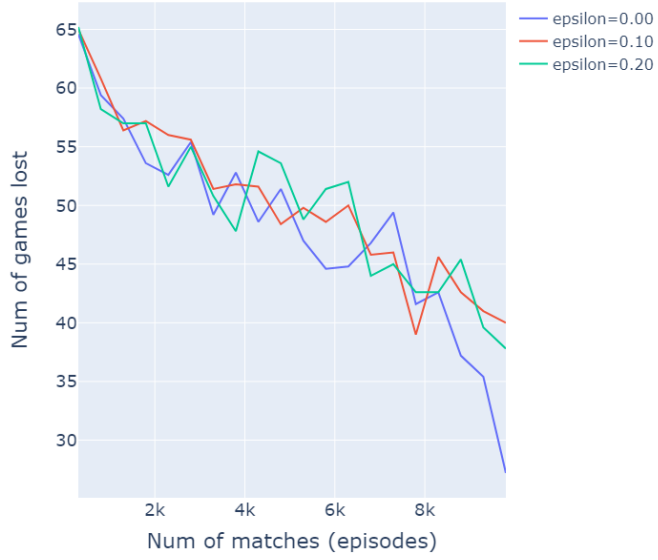
(a) Here we see that for none of the values of ϵ, the average absolute q-value update decreases. This shows that the agent hasn't converged yet.



(b) Again, just like in the experiments with α, all ϵ values explore afterstates at a similar rate. The number of unique afterstates as player 2 is close to 40k.



(c) The same comment mentioned about reward structure vs avoiding losses applies here. All values of ϵ i.e. $0.0, 0.1, 0.2$ work well for this. As many games result in a draw, the avg number of moves is very close to 7.



(d) Number of games lost (per 100 games played). Here, even though loss-rate is about the same for all ϵ, $\epsilon = 0$ has the least loss rate at the end. Again, as mentioned previously, this maybe because of the particular reward structure chosen. It drives exploration despite a greedy action-selection policy.

2.1.3 Results

From the graphs and further discussions (see captions of graphs) we can conclude that:

- For $r = 2$ we want to prefer a lower value of α. This is because - the average absolute q-value update is less for lower α. Also, higher values of α may make large updates in the q-value late in the training.

- For $r = 2$ we want to prefer a lower value of ϵ, even though the average absolute q-value update is similar for all ϵ. This is because the number of games lost suggests that agents with higher ϵ haven't learnt much from the training. We still do not want to set $\epsilon = 0$ as it would make the algorithm an On-Policy Agent (and QLearners are supposed to be Off-Policy).

- For $r = 3$ we want to prefer a lower value of α for similar reasons as stated above (pt 1)

- For $r = 3$ we have tried only low values of ϵ. With this, and keeping in mind that we don't want to set $\epsilon = 0$, we go with $\epsilon = 0.1$

From these results, we will set $\alpha = 0.2$ and $\epsilon = 0.1$ for our further experiments.

2.2 Are afterstates useful?

Yes afterstates will be useful as several state-action pairs map to the same afterstate. This will also help quicken learning as we do not keep separate estimates for different state-actions leading to the same afterstate. But in practice, we do not see the expected 5X reduction in number of estimated values. This is because most of the state-actions haven't been / will not be encountered.

We implement this by observing a simple fact. A state-action pair (s, a) implies one and only one afterstate s_{aft}. Hence wherever we use (s, a), we can simply replace it with s_{aft}

2.3 Before modification ($r = 6,\ n = 25 - 400$)- Discussion only

2.3.1 Estimate number of states

We first make a quick estimate on the number of unique states in the game. The *6 by 5* board is a reduced form of the actual *6 by 7* board. Referring to Wikipedia for Connect-4 gives us $4,531,985,219,092$ unique positions for the full board. Assuming the number of states to be exponential in the number of columns, we get, for the reduced board:

$$(\sqrt[7]{4,531,985,219,092})^5 \approx 10^9 \tag{2}$$

10^9 states! As the original Q-Learning algorithm uses *state-action* pairs, this leads us to an upper bound of 5 billion state-action pairs.

2.3.2 Our specific problem statement

There are 2 arguments against Q-Learning exploring so many states given the particular question. We discuss these points now:

1. **The MC_{200} agent plays first** - This alleviates the problem a little bit. If Q-Learning was the first player, it would have a larger number of states / state-actions to consider. But, this reduction is for 1 turn only. I am guessing an optimistic reduction will be 5X at maximum. Also, the fact that Q-Learning has to estimate values only for its moves, we get another 2X reduction.

2. **The MC_{200} agent will take the same 1 or 2 actions for a given state** - Even despite this, the Q-Learning agent takes all even-plies, and will generate an exponentially-increasing number of states. (due to the ϵ - greedy policy choosing states at random sometimes)

Thus in the end, even though the number of states / state-actions explored are less than stated above, they will still be a large number.

This is seen during the implementation of the *MCTS vs Q-Learning* games. We keep track of the new state-actions / afterstates seen by the Q-Learning agent. Even after training, the Q-Learning agent is still actively finding new state-actions / afterstates.

Taking this into consideration, we see that given the time / no. of games played constraints, it is very difficult to get convergence for any combination of values of ϵ and α. γ which is the discounting factor is always set to 1. This is because there are clearly defined finite-episodes.

3 Question 3 - MC_{0-25} vs Q-Learning

3.1 After modification ($max\ r = 3, n = 0 - 25$)

In order to get Q-Learning to beat MC_{0-25}, I tried 2 strategies and have listed them here:

1. **Training Q-Learning against MC_{25}** - This is the most obvious approach. A Q-Learning agent which is able to beat the strongest agent, should surely be able to beat weaker agents.

   This method does show some promise, but suffers from a setback. MC_{25}, takes some time to play a move ($\approx$ 30 milli secs per move). Although this does not seem much, playing 20k games on a 3X5 board gets us to ($20k \times 7 \times 30m = 4200secs > 1hr$). Consequently, many episodes cannot be played. The Q-Learning agent does not explore all afterstates by end of training. It is surprising that despite this, the Q-Learning agent wins a significant number of games (atleast against MC_{10}, MC_{25} and MC_{50}. And manages to draw most of the games.

   For the experiment, we train Q-Learning with $\alpha = 0.2$ and $\epsilon = 0.1$ against MC_{25} for 20k games. After this, we evaluated the Q-Learning agent against MC_n agents (where $n = 10, 25, 50, 100, 200$). Results are shown in following figures.

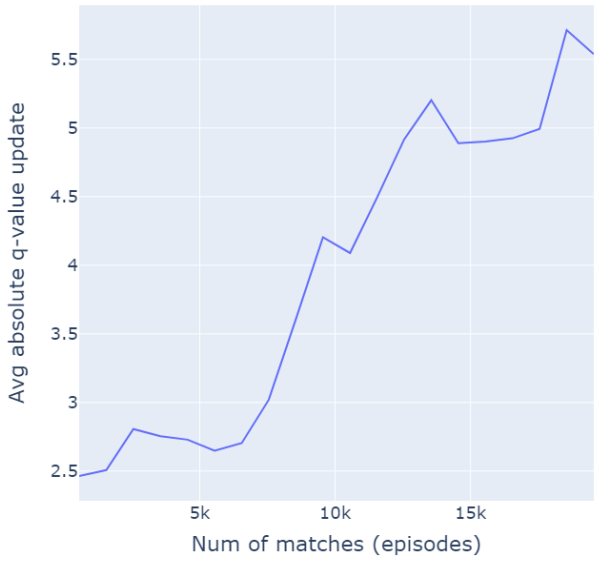
   Time taken for the entire run - Given that most games result in a draw, we take an average of 7 moves per player per game. Thus, time taken is $20k \times 7 \times 30ms\ (Training) + 100 \times 7 \times (15+30+60+110+240)ms\ (Testing) \approx 1hr\ 15mins$. This does not include time it takes to repeatedly load and store data files.

2. **Q-Learning against Q-Learning** - This approach is a very interesting one. Both agents start out with no experience of the game, and play against each other. This process slowly improves both the agents. We will not be able to assess their improvement against each other as they will show similar win-lose rates throughout the training.

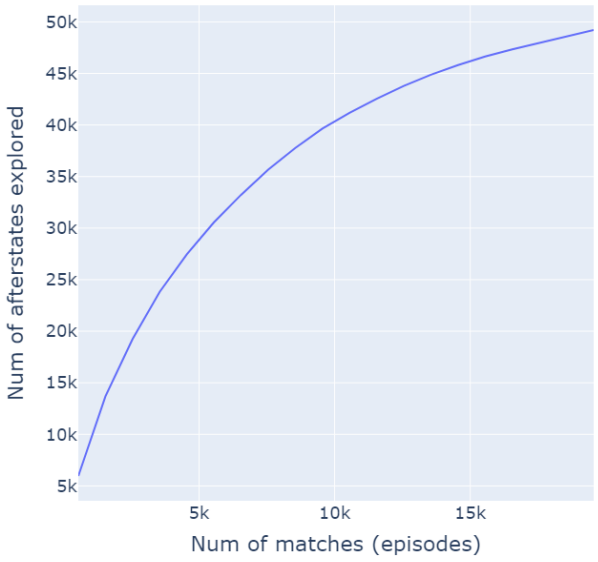
   As the Q-Learning agents choose moves very quickly, I was able to simulate $500k$ games. Both Q-Learning agents have $\alpha = 0.2$ and $\epsilon = 0.1$. Every $50k$ games, we pause, and use the current q-value estimates to evaluate Q-Learning against MC_n agents (where $n = 10, 25, 50, 100, 200$). Results are shown in following figures

Time taken for the entire run - Given that most games result in a draw, we take an average of 7 moves per player per game. Thus, time taken is $500k \times 7 \times 700\mu s\ (Training) + 100 \times 7 \times (15 + 30 + 60 + 110 + 240)ms\ (Testing) \approx 45\ mins$. This does not include time it takes to repeatedly load and store data files.

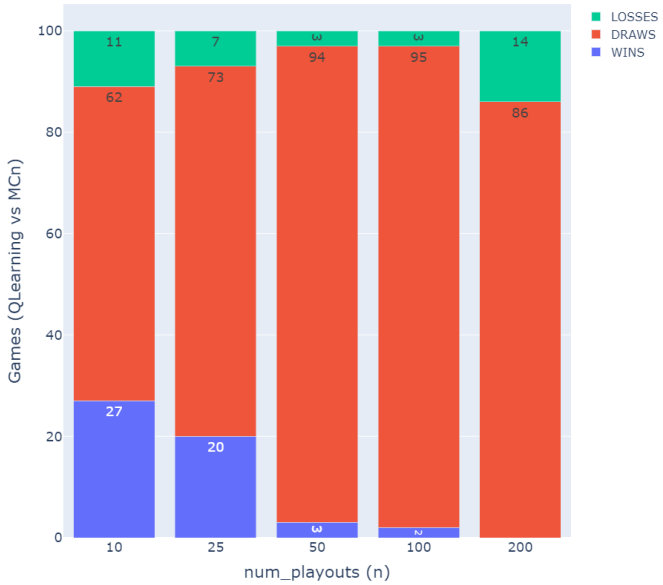
Figure 8: These graphs show the training and testing phases while playing MC_{25} (player 1) against $Q - Learning$ (player 2) on a 3X5 board.



(a) Here we see that even after 20k iterations, the average absolute q-value update doesn't decrease (instead it increases significantly). Thus, there is no convergence. Nevertheless, the Q-Learning agent manages to learn how to win games as seen later

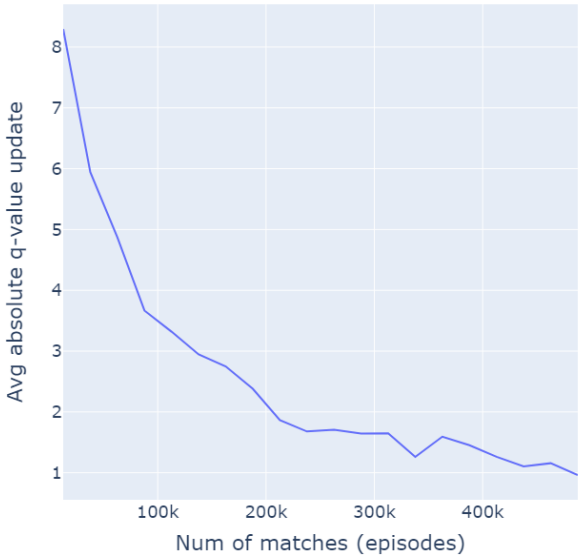


(b) The number of unique afterstates encountered wrt matches played.

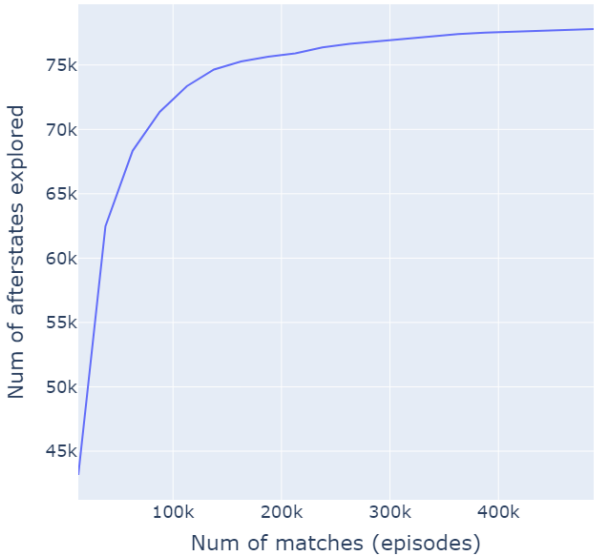


(c) Summary of Q-Learning's games against various MC_n agents. This was captured after the 20k training episodes, by setting $\epsilon = 0$. We can see that most of the games result in draws. Moreover, for $n = 10,\ 25$, the Q-Learning agent wins more games than MC_n does.

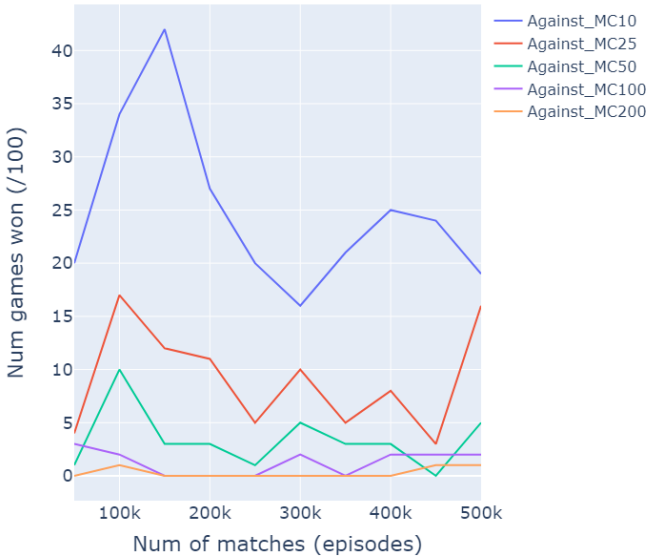
Figure 9: These graphs show the training and testing phases while playing a Q-Learning agent (as player 1) against another Q-Learning agent (as player 2) on a 3X5 board. The testing is done using the value estimations of player 2 (MC_n is player 1)



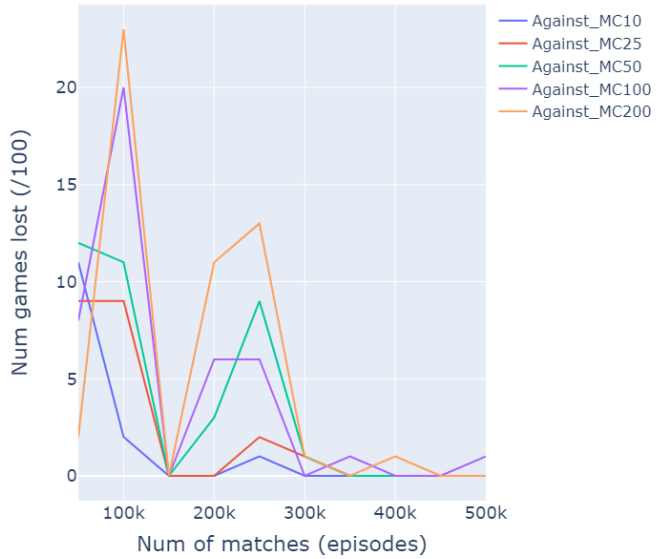
(a) The average absolute q-value update while training Q-Learning against Q-Learning (graph for player 2)



(b) The number of unique afterstates visited wrt number of matches played. We can see here that for a 3X5 board, the number of unique afterstates is close to 80k. This is probably why the training against MC_n agents did not converge, as they were stopped after exploring only 40k-50k afterstates.



(c) Number of games won (per 100 games played). The first data-point is captured after 50k iterations, and hence shows a high win-rate against some agents from the very beginning. The win rate against MC_{10} fluctuates more than others. The overall picture makes sense when seen with the number of games lost(next).



(d) Number of games lost (per 100 games played). We can see that although it fluctuates in the beginning, it decreases significantly and approaches 0 towards the end. After 500k games, it loses only to MC_{100} or MC_{200} (and that too only 1 game in 100)

3.2 Results

We now compare the two training methods and find out which training method results in a better Q-Learning agent.

1. **Q-Learning against $\mathbf{MC_{25}}$** - Here we see that the Q-Learning agent does not converge, yet it learns how to win games against weaker MC_n agents. Hence this method is useful in training Q-Learning agents.
2. **Q-Learning against Q-Learning** - Taking advantage of the fact that Q-Learning agents choose moves very quickly, we could train this setup for much longer than what we could afford with Q-Learning against MC_n (500k episodes vs 20k episodes). Convergence is seen clearly. Although the win rates are similar (or a bit worse) compared to training vs MC_n, we see that the number of losses against MC_n agents is significantly lower here. Because of this, I would consider this is a superior method of training.

3.3 Before modification ($r = 6, n = 25 - 400$) - Discussion only

In order to get Q-Learning to beat MC_{25-400}, I tried 3 different strategies and have listed them here:

1. **Training Q-Learning against $\mathbf{MC_{400}}$** - Unfortunately, given that MC_{400} takes a long time to play, we cannot simulate many games. In an experiment, only 1000 episodes were tried. Consequently, Q-Learning agent does not converge, and does not play very well (even against MC_{25}). Graphs are not shown.
2. **Q-Learning against increasingly strong $\mathbf{MC_n}$ agents** - Because stronger MC_n agents (larger n) take more time to make moves, it will be very time-consuming to run thousands of episodes. The strategy is that after learning to beat weaker MC_n agents, lesser training will be required to beat stronger agents.

   This method also didn't work very well, but was able to win some games against MC_{20} and performance was better than Q-Learning against MC_{400}. Graphs not shown.
3. **Q-Learning against Q-Learning** - Same approach as used "after modification". Unfortunately, here even though we can play many games between the agents, the Q-Learners do

not converge (recorded $600k$ plus unique afterstates each). The performance here is similar to training Q-Learning against increasingly strong MC_n agents.