Mastering Adversarial Monte Carlo Tree Search using Q-learning RL agent on Connect-4

Mehul Gupta

2019A7PS0033G

BITS Pilani K K Birla Goa Campus

CS-F407 Programming Assignment-2 Report

Abstract

This report implements a version of Adversarial Monte Carlo Tree Search to play the game of Connect-4 and simulates multiple games with MCTS agents playing against each other to provide insights about the various parameters involved in MCTS algorithm and their effect on the agent~~s~~ performance. Further, I go on to implement a Reinforcement-learning based agent that learns to play Connect-4 against MCTS agents using Q-Learning algorithm and study games played between the two agents to evaluate the performance and gain insights about the nature of the two algorithms.

Overview

This report first explains the exact problem specification, then goes on to give an introduction about the Connect-4 game and the two algorithms, Monte Carlo Tree Search and Q-learning, implemented in this report to play the game and the ideas behind various changes and modifications to the generic algorithm for each. Next, various experiments are performed by simulating games between different versions of the agents of the algorithms mentioned above and the effect of every change is logged and graphed to provide insights about the working of the algorithm. Finally, we conclude this report with some closing remarks about the algorithms employed and the insights gained.

Problem Specification

We have to implement an adversarial Monte Carlo Tree Search algorithm that is capable of playing the game Connect-4. Further, we have to create a Reinforcement-learning based agent that uses Q-learning to learn to play against the MCTS algorithm designed above with the objective to maximize the number of wins and minimize losses over the games played with Monte Carlo Tree Search.

Connect-4: Game description

Connect 4 is a two-player connection board game, in which each player chooses a color and the objective is to take turns dropping colored discs into m-columns with vertically arranged n-rows which go on to occupy the lowest available space within the column, so as to form a horizontal, vertical or diagonal line of four of ones own disk. In this report, a Connect-4 board of (5 columns x 6 rows) has been used to simulate MCTS agents and the Q-learning algorithm is trained to play on a Connect-4 board of (5 columns x 4 rows) against MCTS(n) agents.

Monte Carlo Tree Search Algorithm

Monte Carlo Tree Search algorithm is based on the idea to estimate the expected utility of a state through multiple playouts of the game from the starting configuration and using it to make its next move. The algorithm achieves this by using a playout policy that selects and explores states with higher estimated utility to arrive on an estimate for the current state, relying on the fact that states with higher utility should have a higher probability to be taken so as to maximize the final reward. It maintains all the information about states using a tree data structure

The basic Algorithm performs multiple playouts to estimate the root state each of which includes the following steps:

1. **Selection**: It uses a selection policy based on UCB (Upper Confidence Bound) to select states with higher utility or states less explored to decide the next best move to make, and keeps simulating moves and traversing the Monte Carlo tree correspondingly in parallel until it reaches a leaf node.

2. **Expansion**: Expands the selected leaf node to expose children states that can be reached through the current state and selects one.

3. **Simulation**: Next the algorithm simulates the game until it terminates to estimate the reward associated with the leaf node.

4. **Back-Propagation**: The estimated reward of the leaf state is backpropagated through all the nodes in the path traversed during the playout and all their stats are updated.

This is repeated multiple times to arrive at an estimate for the current state/node. The number of playouts is usually fixed for an agent. In the report, whenever the term MCTS(n) is mentioned, it refers to an MCTS agent that performs n playouts. The stats obtained through these simulations are used to select the next move to be taken by the agent. The idea is to use the most explored child state of the current state as the playout policy dictates that the node with the highest utility was the one picked most often.

Since an MCTS agent implemented to play Connect 4 has to be adversarial in nature, certain modifications had to be made so as to enable it to perform in this scenario. This involved modifying reward of terminal states to be 1 or -1 for a win/loss respectively and 0 for a draw, performing the algorithms mentioned in the steps above in a min-max fashion so that the agent can model how a rival agent will choose the next move during its turn so as to achieve an accurate measure of the expected utility of moves. For the first move, the game tree is constrained to a depth of 4.

After the move is taken by either players, the MCTS algorithm updates its tree and the root node based on the child state taken. This allows the agent to utilize previous estimations of states so as to enable a more accurate measurement of expected utilities and play better rather than start from scratch.

Q-Learning

Reinforcement-based learning agents are algorithms that learn to play the game using multiple simulations based on the sole aim of optimizing the rewards out of the game. Q-learning is one such RL based algorithm that we have used to learn how to play against MCTS(n) agents.

The basic idea behind these models are that they model the environment using a Markov Decision Process with the property that a state can be modelled solely on the basis of its next state. Q-learning is an off-policy TD learning algorithm that maintains and uses the Q(state,action) value (Expected Reward of a state, action pair given the agent thereafter follows the optimal Q-policy) to learn how to play. The above mentioned policy is used and the agent is made to play multiple games to converge on appropriate Q(s,a) values using the update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big].$$

For this report, the Q-learning agent has been trained on the Connect-4 board of 4x5 against MCTS(n) agents with n varying from 0 to 25. Rewards for states have been set so as to increase the number of wins, discourage stalemates slightly, and reduce the number of moves. Win terminal states have been rewarded by 100 points, losses by -100 points, and stalemates by -50 points, and transient states by -1. Stalemates and longer games are penalized to discourage those.

Q-values have been stored in a lookup table fashion, using the concept of after-states as keys to look up the desired Q-value for a (state, action) pair. The idea is implemented as follows, whenever we need to look up the Q-value of a (state, action) pair, we produce the next state by performing the action on the current state, then the expected rewards (aka the after-state value) of the next state is used as the Q-value for the given pair. The validity of the idea can be seen from the following argument, in a game like Connect-4, there are multiple (state, action) pairs that produce the same after-state and thus should have the same Q-value as the for any such pair, the immediate state produced by the action is known and the only uncertainty is regarding how the opponent will react. In a sense, for such games, the value of actions is defined by their immediate effects. Thus the after-state values and the Q(s, a) values should represent the same thing. Therefore, since by encoding (state, action) pair by a single (next_state), we can greatly reduce the size of the lookup table and converge faster to the optimal solution.

Experiments and Empirical Observations

1. Monte Carlo Tree Search analysis

To study the performance of MCTS agents over different parameters, the following experiment was devised.

Two versions of the Monte Carlo Tree Search (MCTS) agents are created to play Connect-4 game on a board of (5 columns x 6 rows) against each. The first agent is an MCTS(40) agent that performs 40 simulations before choosing an action, and the second agent MCTS(200) agent performs 200 simulations before choosing an action. The value of each state is governed by the total reward over the game. The two agents are made to play against each other for 100 games, with each agent being the first player for 50 games to remove any bias caused by any possible first movers advantage.

The most important parameter for an MCTS algorithm is the parameter C which is used in the UCB estimation of states during the selection policy.

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

To analyze the performance of MCTS(n) agent, this experiment was run over a wide variety of values for C sampled from 10^{-2} to 1000 and logarithmically spaced over the entire range. The graph below summarizes the performance of the agents during the experiment.

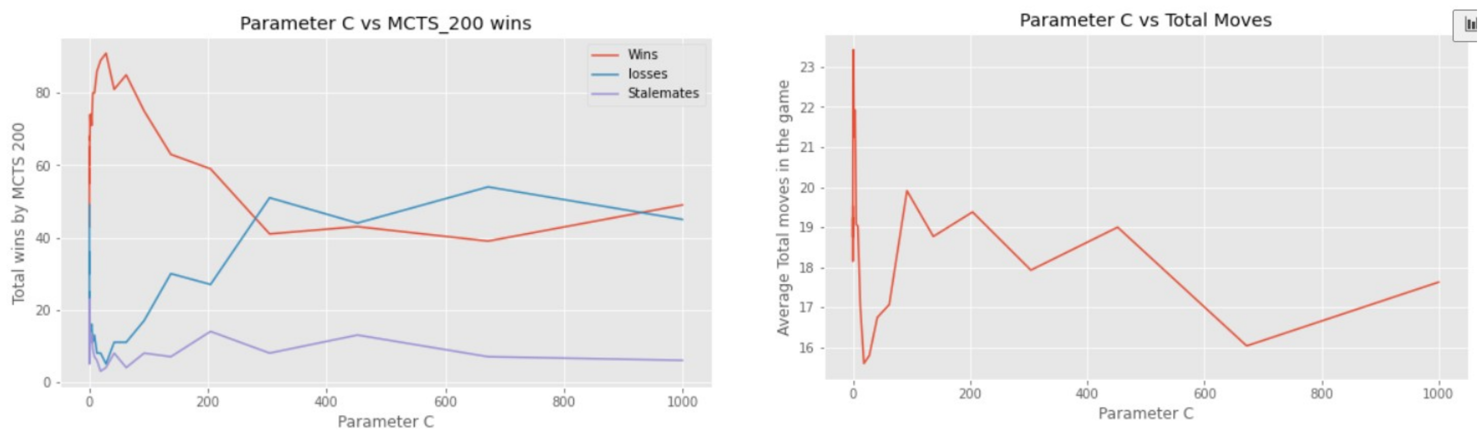


Fig 1: Performance of MCTS_200 vs MCTS_40 over different values of C

As can be seen from the above graphs, the general trend can be observed as follows: For initial values of C, the wins attained by MCTS_200 increases till it peaks by around C=50, after which it starts to decline till about C=300, after which both the agents perform equally. On observing the graph capturing the total moves made during the game, we can see that the total moves is high for initial values of C, after which it strongly dips in the region from C=10 to C= 50 to about 16 moves on average, after which it rises again.

From the above graphs, we can note the dominant performance of MCTS_200 over MCTS_40 as the wins for most part dominate the loss percentage. This is to be expected as MCTS_200 performs 200 simulations before choosing the next move as compared to 40 simulations by MCTS_40, hence it is more likely to converge on better estimates for the utility of each state during the game hence make better moves.

As can be seen from the equation for UCB function, the parameter C acts as a control over the tradeoff between exploration and exploitation during the selection policy of the MCTS algorithm, with higher values of C favoring exploration over exploitation of states with better estimated utility. The poor performance of the MCTS_200 agent (higher total moves and lower win ratios) for values of C below 1 can be attributed to lack of enough exploration to discover states and actions with better expected rewards. The performance then goes on to peak by around C=50, where the tradeoff between exploration and exploitation seems to be balanced more for MCTS_200 such that MCTS_200 manages to win more than 90% of the games and finish the game in approximately 16 moves while the MCTS_40 due to the low number of playouts available because of which the term multiplied to C does not manage to drop down to values low enough that it can show signs of exploitation. Thereon the performance worsens till about C=300, after which the performance of both MCTS_200 agent and MCTS_40 agent seems to be approximately the same. This phenomenon can be explained by the fact that at such high values of C, the agent is more driven to highly explore instead of exploiting the estimated utility of states to achieve better performance, and by around C=300, the agents are almost randomly choosing states instead of taking the better next-states.

To further study the effect of C on an agents performance, the same experiment as above is repeated but with the following change, both the agents used are MCTS_200, but for player1, C is sampled from 10^{-2} to 1000 and logarithmically spaced over the entire range, while the C for player 2 is fixed at 2. Now the benchmark is performed which produces results as follows:

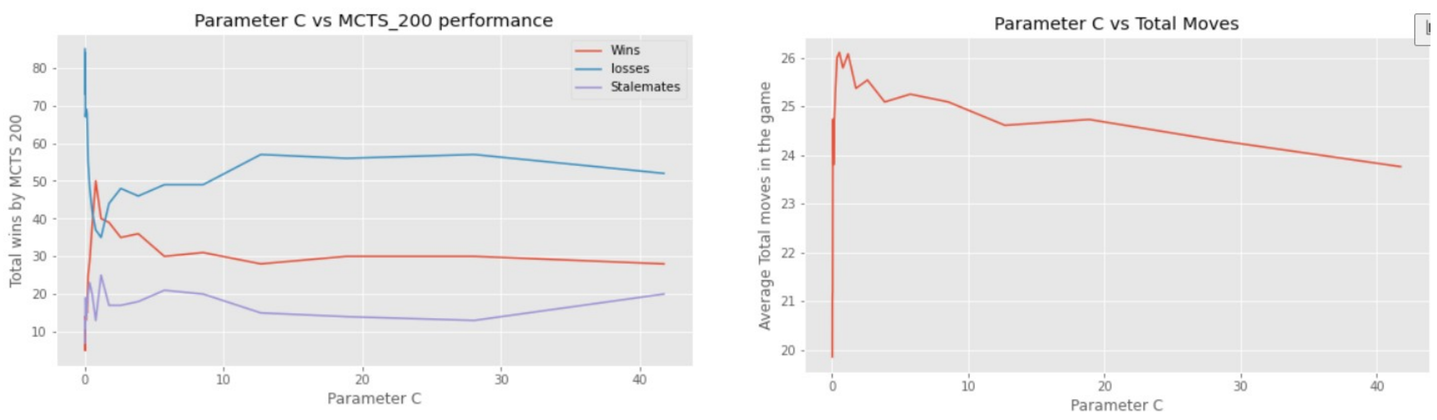


Fig 2: Performance of MCTS_200 over different values of C

The above graphs give a more direct relationship between the parameter C and the performance of the agent. Most trends observed can be explained by the ideas mentioned before such as poor performance for C below 1 and large values of C. The most optimal value lies in the C = [1,2] range, with C = 1 giving the best performance and even outperforming player 2 and winning more games. Note that even at the most optimal value of C, the player 1 MCTS agent only manages to win around half the games because the player 2 is initialized with C=2 which is also a strong optimal parameter candidate. The high number of total moves for C=[1,2] can also be explained by this as around this range, both the agents are equally matched and the player 2 does not manage to end the game in smaller number of moves.

2. Evaluation of Q-learning over different hyper-parameters

Next, I went on to implement an RL-based Q-learning algorithm that plays against an MCTS_n agent for n that can vary anywhere from 0 to 25, with C fixed at C=1 (found from the previous experiment). The experiments are designed as follows:

The Connect4 game is played on a (5 columns x 4 rows) board. The game starts with MCTS_n agent making the first move and then the Q-agent and so on till the game terminates. A training loop runs for a fixed amount of epochs (200). Inside each epoch, an MCTS agent is generated with the parameter n randomly chosen between 0 to 25, after which the Q-agent plays batch_epoch (10) number of games with the MCTS_n agent and learns from it. The performance logged over the entire simulation is used to analyze the performance of the Q-agent over various hyper-parameters.

First, we analyze the performance of Q-learning algorithm over various hyper-parameters by graphing the relationship between total wins over the sampled parameters for some important hyper-parameters including the alpha learning rate constant and the gamma discounting factor constant used in the Q-learning update step. The control values used for the Q-learning algorithm were:

- alpha: 0.1
- gamma: 0.4
- greedy_prob: 0.05 [The probability of choosing a random action]

The graphs for the experiments are shown below.

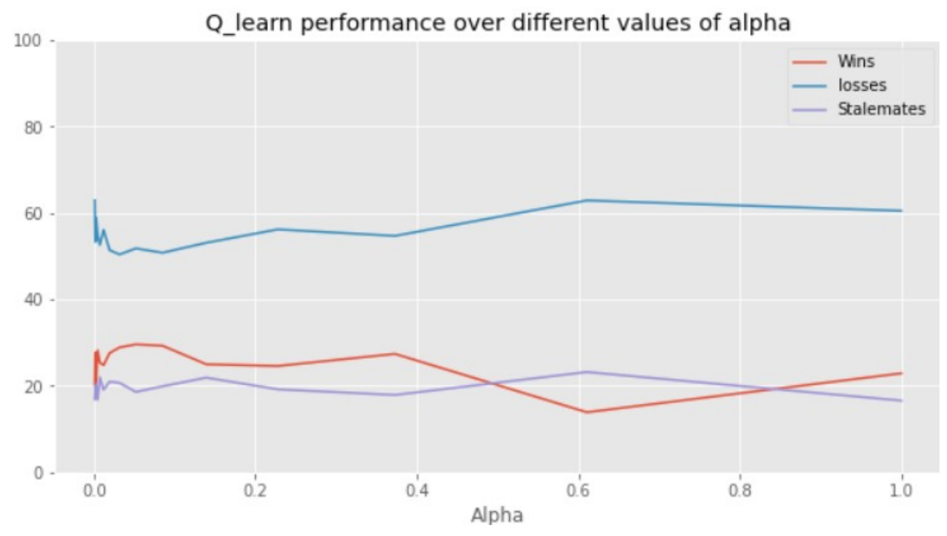


Fig 3: Q-learning performance over different values of alpha

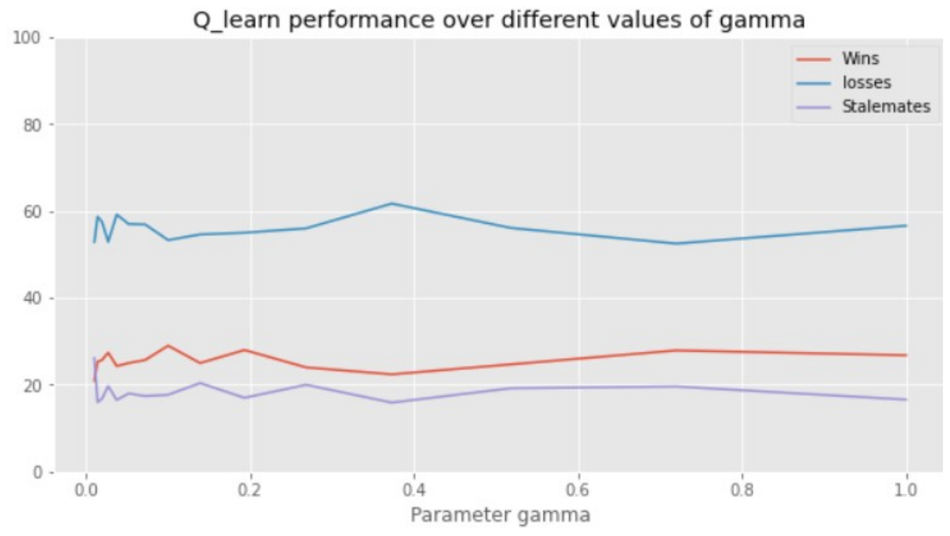


Fig 4: Q-learning performance over different values of gamma

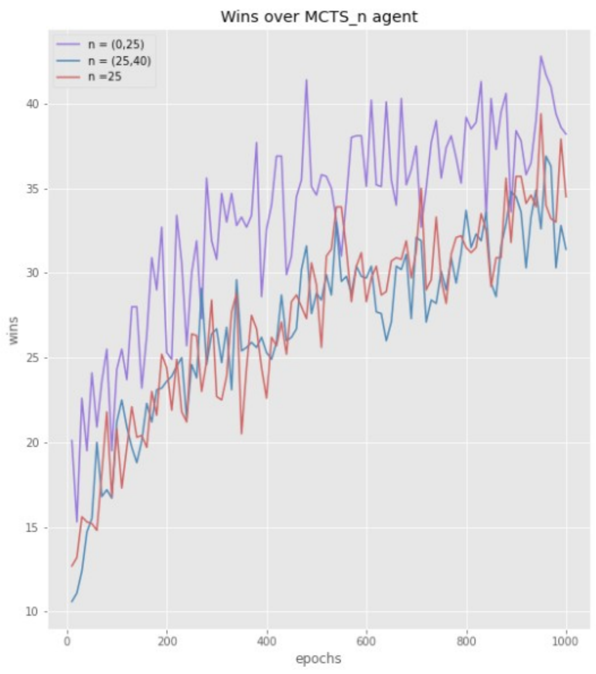
The training performance seems to be almost similar over the range, the most optimal performance at alpha around 0.07 and gamma around 0.7.

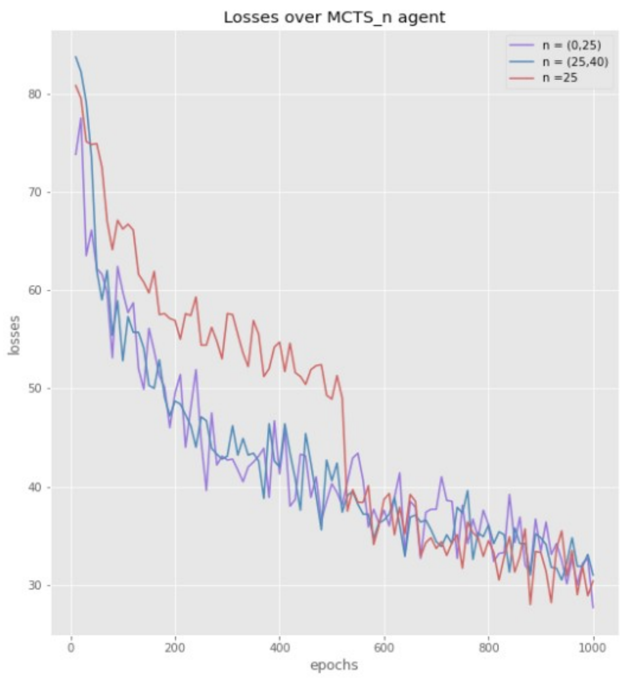
Next, we design 3 different methods to train the Q-agent to play against an MCTS_n (n = [0,25]) agent and analyze the performance of the 3 different techniques to choose one that converges the fastest.

The experiments is designed as before with some minor changes, the optimal value for the hyper-parameters obtained from above is used. After every 10 epoch, we test the performance of the Q-agent learned against an MCTS agent of n = [0,25], this value is logged and later graphed to provide insights about the performance of the different Q-learning techniques employed, which are as follows:

1. Technique 1: The Q-agent is trained against an MCTS(n) agent with n = [0,25]
2. Technique 2: The Q-agent is trained against an MCTS(n) agent with n = [25,40]
3. Technique 3: The Q-agent is trained against an MCTS(n) agent with n=25

The training performance observed is as follows:





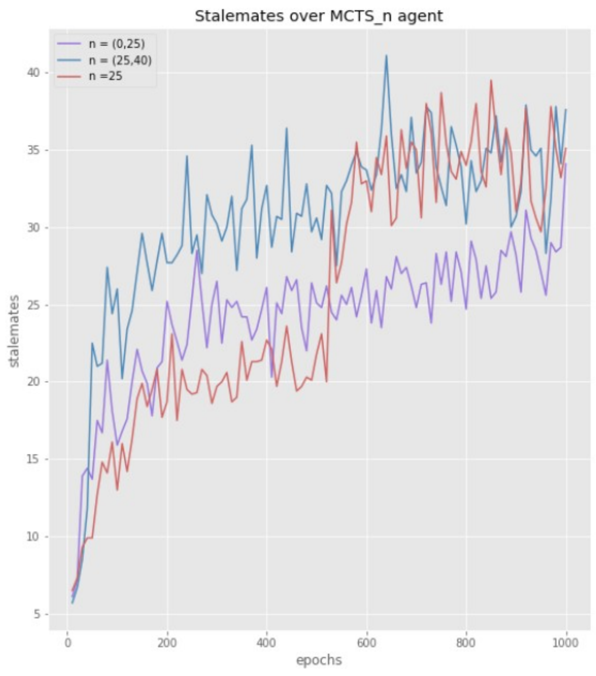


Fig 5: Convergence of different Q-learning training methods over epochs

From the above graphs, we can infer that Technique 1, training the Q-agent against the same adversary MCTS(n) for n=[0,25] results in faster convergence to optimal Q-values and thus results in better performing agent.

Now we finally have all the ingredients to train a Q-agent to play Connect-4 against an MCTS agent.

3. Training a Q-learning agent

Finally, we design the training process to train a Q-learning agent as follows.
The Connect 4 board chosen is of dimensions (5 columns x 4 rows). The MCTS_n agent used to train the Q_agent is initialized with the C parameter = 1 with n varying from 0 to 25.

The Q-agent is designed as mentioned in the introduction and initialized with the hyper-parameters alpha = 0.07, gamma = 0.7, greedy_prob = 0.05. MCTS_n agent is the first player and Q-agent is the second player. Now the training is performed for 10,000 epochs, in each of which, an MCTS_n agent is generated and plays for a batch_epoch = 10 games.

The graph summarizing the training process is as follows:

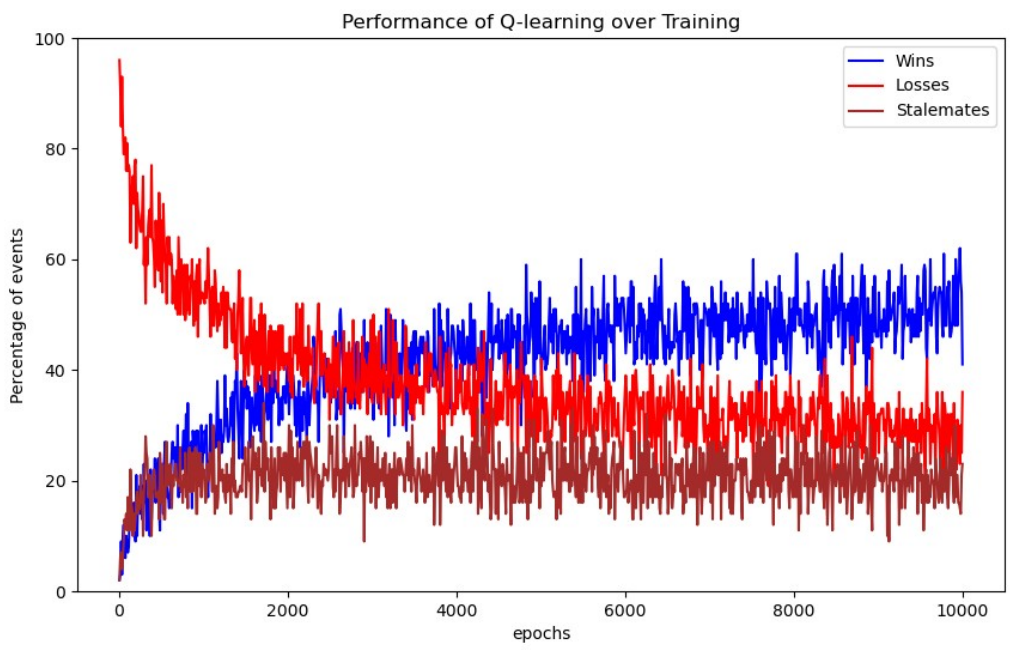
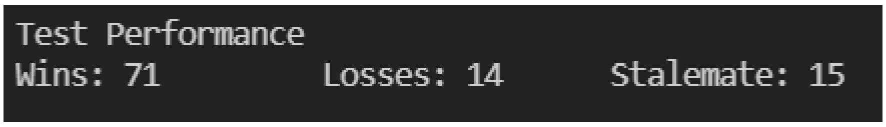


Fig 6: Q-learning training performance

Finally we test the performance of the final Q-learning agent model produced on MCTS(n) agent with n varying from 0 to 25:

As we can see, we have managed to achieve a win percentages of 71% and drawn in 15% games, thus pushing the loss percentage to 14%.



```
Test Performance
Wins: 71        Losses: 14      Stalemate: 15
```

Conclusion

Monte Carlo Tree Search and Q-learning Reinforcement-learning based algorithms are both powerful algorithms that can be used to play a variety of games from Connect-4 to GO etc. We have demonstrated the performance and effect of various parameters and modifications to the algorithms and ideas about how to go about selecting the most appropriate model for our game.

Monte Carlo Tree Search is a highly versatile algorithm and can explore games to extremely high depths even in games where the branching factor is high as compared to minimax algorithms that require huge amounts of calculations and memory for medium amount of branch factors and can usually not be extended beyond depth 6 or 7. While this report demonstrates the performance of Monte Carlo Trees for a simplified version of Connect4 (5 columns x 6 rows) due to memory and system constraints, this algorithm has been deployed along with deep learning RL techniques to create models like Alpha GO and Alpha GO Zero which can play increasingly complex games like GO, where these models are simply MCTS algorithms with RL based learning employed during the selection and simulation steps.

Q-learning algorithm is one of the most popular reinforcement based learning algorithms which finds application in the most diverse selection of tasks, from teaching agents to play games to control theory tasks like autonomous driving, teaching robots how to perform simple actions like picking up objects, moving etc. The Q-learning algorithm implemented for this report uses a lookup table for storing the Q(state,action) pair, hence we have restrained the game to a simplified version of Connect 4 board (5 columns x 4 rows) due to memory and system constraints. In research and industry, this lookup table is replaced by deep learning networks that learn a function to map (state, action) pairs to Q-values, and hence is able to be applied in applications with large or even continuous state/action spaces.

References

Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach.
Richard S. Sutton, and Andrew G. Barto - Reinforcement Learning: An Introduction