Reinforcement Learning based CONNECT 4

Tarun Srinivasan
MS Robotics
Northeastern University

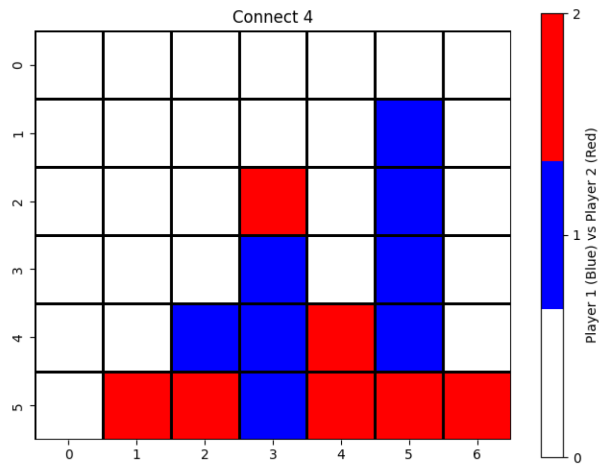
Husain Gittham
MS Robotics
Northeastern University

1 Introduction

For this project, We wanted to build and implement two different Reinforcement Learning algorithms which are fundamentally different for the classic game of Connect 4. The game is pretty simple in the high level, with a 6 x 7 grid space into which we drop coins to try to form a continuous line of 4 coins. It becomes a bit complicated due to the constraints brought to you by the opponent who basically tries to win and also hinder your chances of winning.

The idea behind this project was to implement two or more (if time permits) and evaluate the compromises each algorithm has toward the result. I proceeded to choose one model-free and a model-based approach to cover both aspects of the RL problem. And for this the initial plan was to use **Deep Q Learning** as the model-free algorithm and **Monte Carlo Tree Search (MCTS)** for the model-based approach.

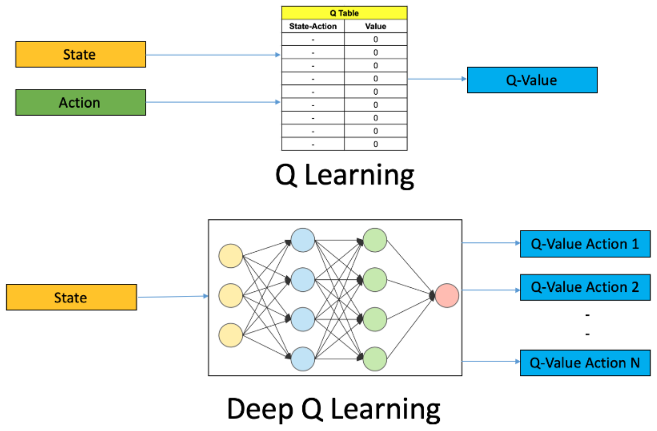
Both the models have been applied to various games with varying state and action spaces. The **exploratory advantages** for MCTS has been recorded across various literature. And the **intelligence** of Deep Learning is a known advantage. So I thought it would be relevant to put them to a test and evaluate them. The evaluation and conclusions drawn can further be used to derive possible future improvements and further scope to find better model formulations for the application.



2 Algorithms

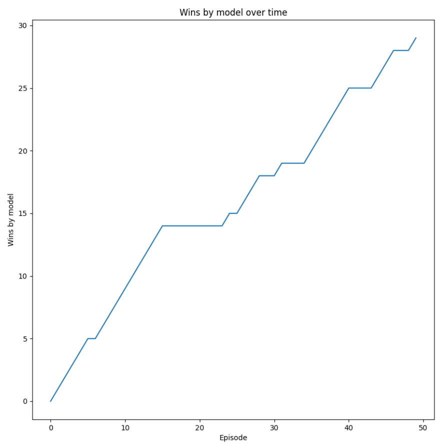
2.1 Deep Q Learning

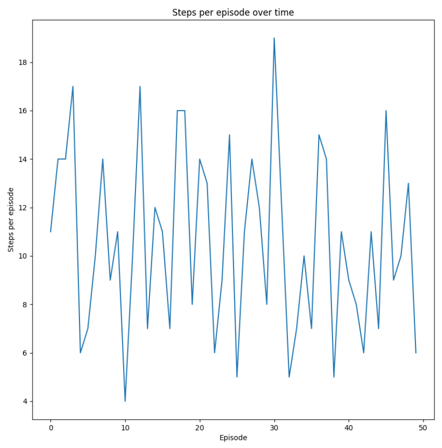
We have seen how Q Learning works in class, in a general sense we try to estimate the Q table to keep track of the situations that can arise and take an action that can maximize rewards. But this approach even though it works well, has a disadvantage here fro our application as the state-action combination for connect 4 is pretty large compared to simple games like discussed in class. According to an encyclopedia for integer combinations [3], there are around 4,531,985,219,092 positions for this game and this complexity makes using Q learning a less attractive option. Especially considering the computational time and efficiency. But this is where Deep Learning excels. The picture below taken from [1] shows the difference and why Deep Q Learning is preferred in this case.



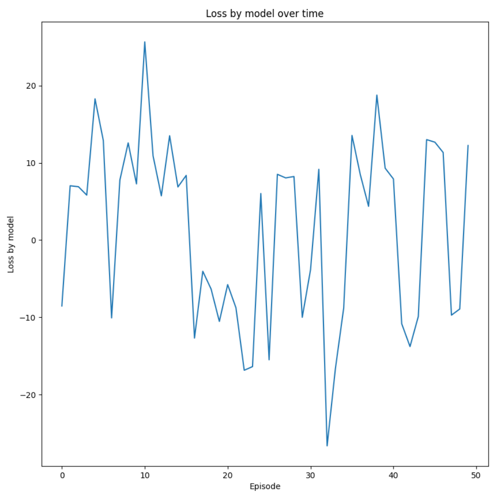
2.1.1 Tensorflow implementation

We initially tried to implement DQNs using Tensorflow, using a Kaggle Environment by default. This did not really turn out the way we expected it to, performance was not good at all to be honest. The loss curve did not show any improvement and the model only won 30 percent of the time out 100 training episodes, with no improvement. The win rate curve was linear as shown below. This showed us that the model had a difficult time learning the necessary.





The loss was also monitored over the training period, which also did not show signs of training. We did not really catch on to why it wasn't able to learn. My initial hypothesis was that it was something to do with the kaggle environment which we did not want to get into and change.



Since we had a short time span, we dumped this and moved on to use a pytorch environment like in the homework.

2.1.2 Pytorch Implementation

We implemented our own class for the environment this time to avoid any issues with kaggle. The pytorch implementation needed a bigger compute, since we were using 256 as the batch size and a bigger model. We used the Discovery Supercomputer cluster at Northeastern to train our implementation with nodes equipped with NVIDIA TESLA V100 SXM2s. This implementation was then used for the rest of the project as it worked best.

2.2 Monte Carlo Tree Search

A well documented approach, The core idea of MCTS [2] is to successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS is generally considered a model-based approach because it builds a search tree to simulate and explore the possible future states and actions based on a model of the environment. This model can be a simplified, learned model of the environment, or an actual simulator.

Monte Carlo Tree Search builds a search tree with n nodes with each node annotated with the win count and the visit count. Initially, the tree starts with a single root node and performs iterations as long as resources are not exhausted.

The entire algorithm in a general overview can be split into four important phases. Selection, Expansion, Simulation and Backup.

Selection — In this phase, the agent starts at the root node, selects the most urgent node, apply the chosen actions, and continue till the terminal state is reached. To select the most urgent node **upper confidence bound** of the nodes is used. The node with the maximum UCB is used as the next node. The UCB process helps overcome **exploration and exploitation** dilemma. Also well known as a Multi-Armed bandit problem where the agent wants to maximize one's gains while playing (Lifelong Learning).

Expansion — When UCB can no longer be applied to find the next node, the game tree is expanded further to include an unexplored child by appending all possible nodes from the leaf node. This is the exploration advantage that we see as compared to other models.

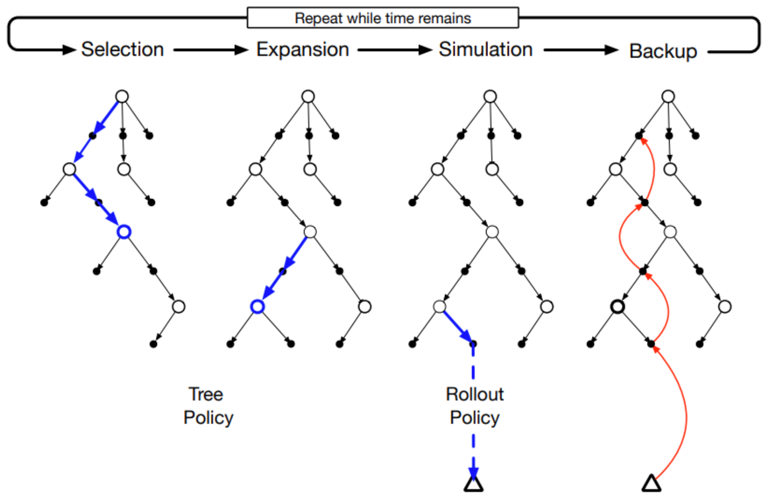


Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

Simulation — Once expanded the algorithm selects the child node either randomly or with a policy until it reaches the final stage of the game

Backpropagation — When the agent reaches the final state of the game with a winner, all the traversed nodes are updated. The visit and win score for each node is updated.

The above steps are repeated until time remains, which for our application we used a iteration counter (MCTS Iterations) and end the loop after it reaches a certain threshold. Finally, the child node of the root node with the highest number of visits is selected as the next action as more the number of visits higher is the UCB for that child node which denotes a certain action basically in our application.

2.2.1 Implementation

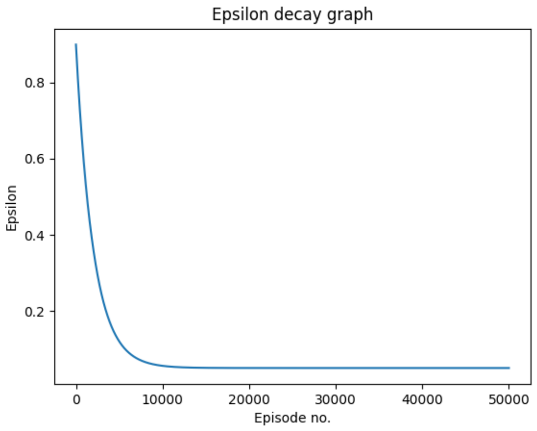
We used the same environment class defined for the Deep Q Learning model. To keep things modular, we use the same functions from that implementation wherever possible, just changing the algorithm. This will make things easy when we try and evaluate the algorithms. This implementation can run on local CPU, but might take a lot of time, due to the tree building and iterative aspect of the algorithm. This algorithm was **tested** on a local GPU with a 6th-generation Intel Core i7-6700HQ 2.6GHz and 16GB RAM.

3 Experiments

3.1 Training the Deep Q Learning model

For training the model, we first trained on smaller episodes to tune the hyperparameters like batch size, model size and epsilon values. The best parameters were chosen and used for the entire experiment. The best epsilon value turned out to be 0.9 but we came across sources on the internet that showed better results on other tasks while using a decaying epsilon as discussed in class. So we implemented a decaying epsilon which decays from 0.9 to 0.05 as the episodes increase. This makes sure that the model explores a lot in the beginning and then exploits a lot during the end. Which is how we want the model to learn.

The plot for the epsilon decay for 20k episodes is given below:

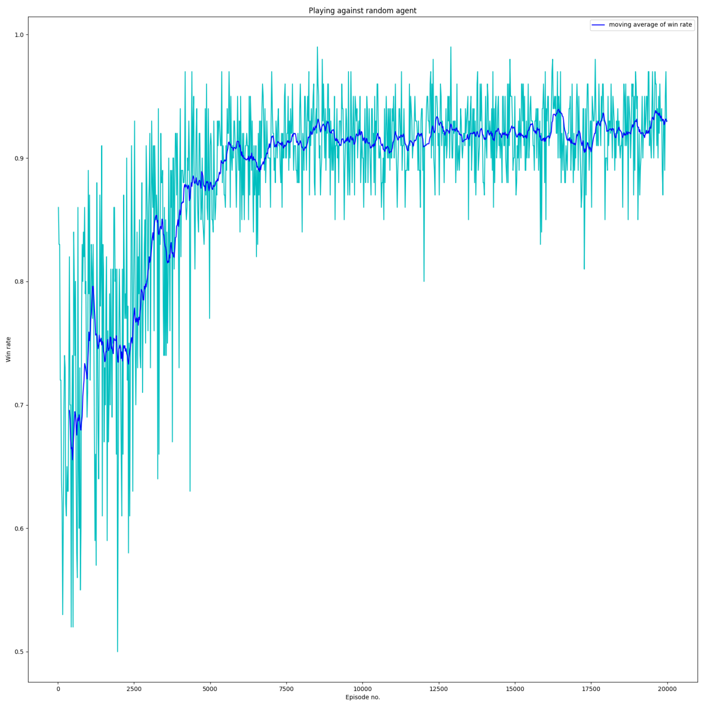


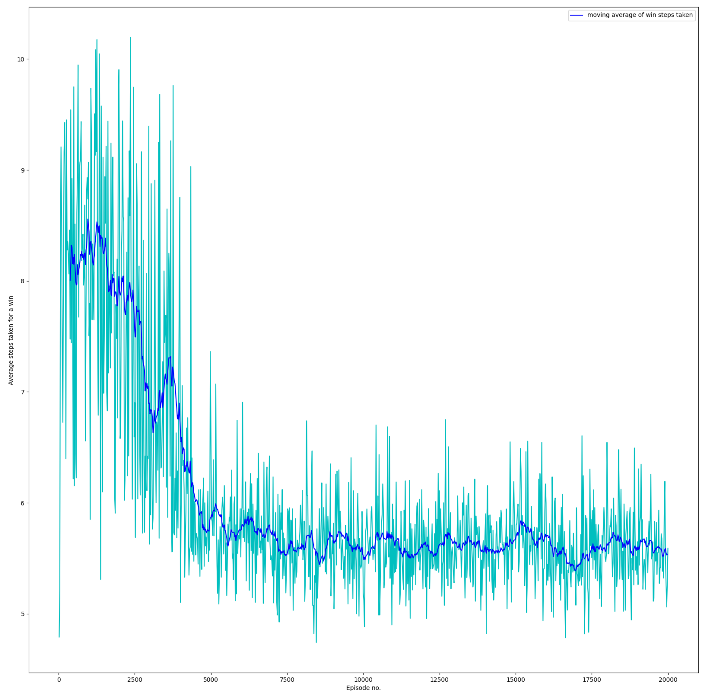
The loss we used was Smooth L1, which is basically a combination of L1 (absolute rms) and L2. But the Smooth L1 gets its name as it converges to the L1 loss. The inputs to the model was the board state, which is a matrix of size 6x7 with zeros, with ones and twos on spaces occupied by each player. Batch size is set to 256 for all experiments.

3.1.1 Training Graphs

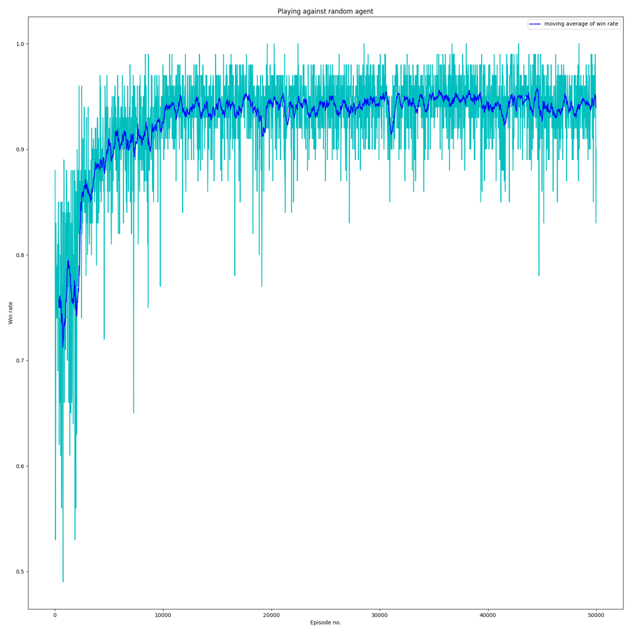
We report findings for two different (best) iterations we had on the Discovery Cluster. One model trained with 20k episodes and the other trained on 50k episodes. This was done to confirm convergence and purely to avoid premature stopping.

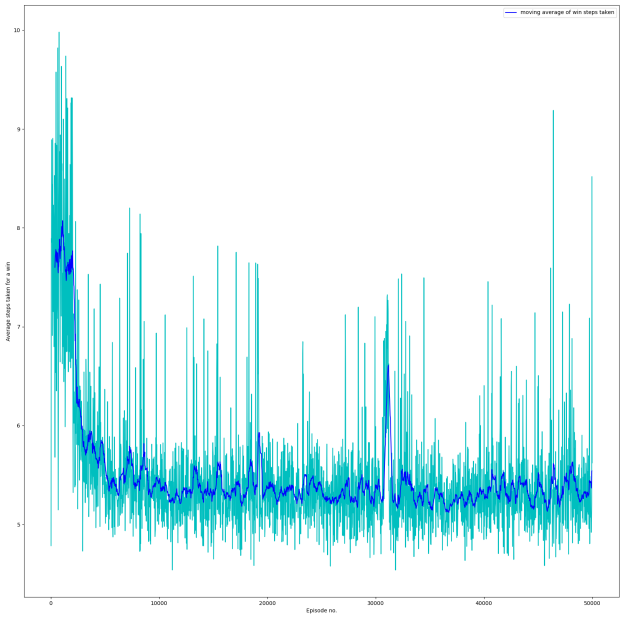
Training results for the model trained on 20k:





Training results for 50k episodes:

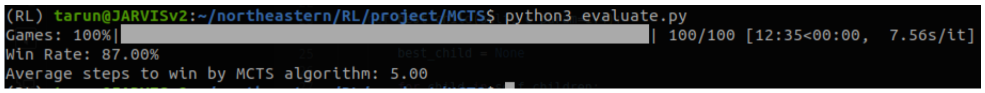




The model was trained against a random opponent. And seeing the 20k episodes, we see that it seems to converge in the upper nineties. I wanted to make sure that we are converging and not ending the training before the model could learn, so we ran another iteration with 50k episodes which showed us that the model actually converged as the graph remained the same apart from being noisy all the same.

3.2 MCTS Experiments

Since MCTS as a basic model does not really learn and save models hence there is no training phase attached to this algorithm. So we jump directly into evaluating the MCTS Algorithm on different hyperparameters, the most prominent one with large effects on the outcome being the iterations we run as mentioned before.

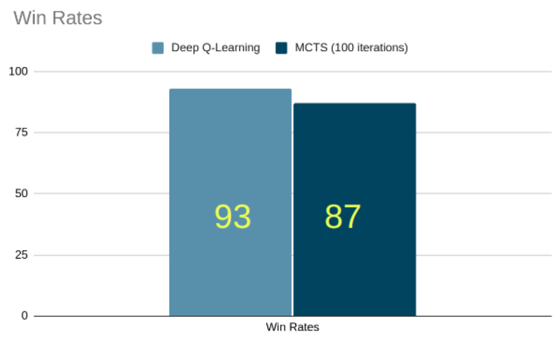


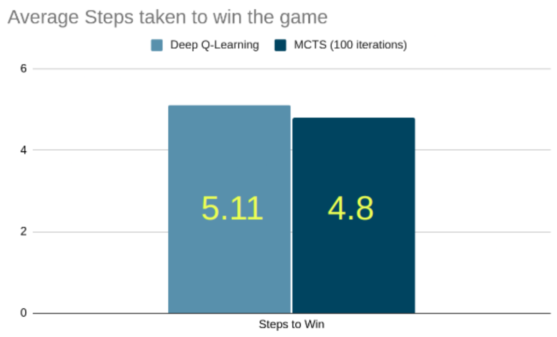
```
(RL) tarun@JARVISv2:~/northeastern/RL/project/MCTS$ python3 evaluate.py
Games: 100%|████████████████████████████████████████████████████████| 100/100 [12:35<00:00,  7.56s/it]
Win Rate: 87.00%
Average steps to win by MCTS algorithm: 5.00
```

The algorithm was able to win against a random opponent starting first, **87 percent** of the time but this value kept changing and averaged at around 87 percent on multiple tries. This showed that 100 iterations was a good option to test out. We also tried playing around with the number of iterations for further testing.

3.3 Comparing evaluations on both algorithms

After training, the Pytorch model was saved and loaded on to a demo environment where 100 games are played and we tested the win rates and average steps taken to win the game to compare both the algorithms.





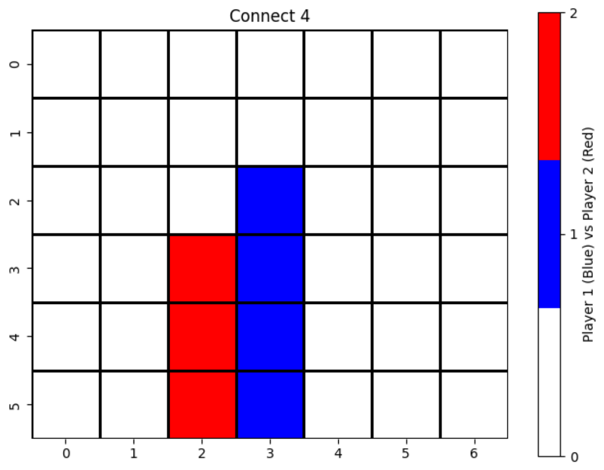
Seeing the comparisons against a random opponent both the models performed really well having no problem beating it. They even took only 5 steps on average to complete the game.

To have a more robust evaluation, and to negate any possibility that the DL model has just learnt to beat the random opponent, we put them against each other and evaluated them.

3.3.1 Problems while evaluating the algorithms against each other

One of the major issues was that I was getting a zero percent win rate on the second player whichever model I chose, initially we thought that this was because the game is solved and the first person can always win, but we tried to render the game boards for each win and noticed the issue.

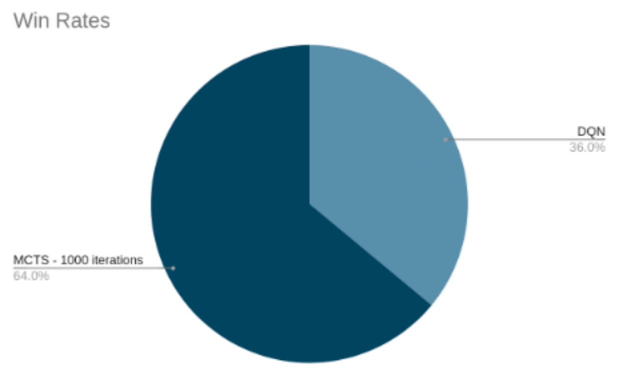
Both the models try to start the game in the middle column, and continue doing the same things. Since MCTS is a tree based method, it comes up with the same possibilities and same scenarios for the next move and the DL model just keeps repeating the same actions to win. Below is the scenario which kept repeating for over 100 times.

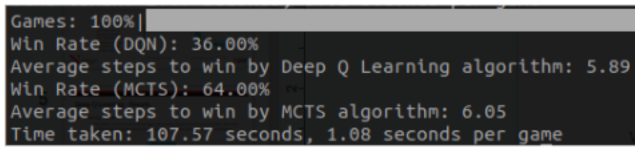


To deal with this situation, we came with a solution, to start the game in a random setting. Let the random agent play the first moves for each of the AI algorithms and then let the models take over. This brought in enough entropy into the problem to actually test the models efficiency.

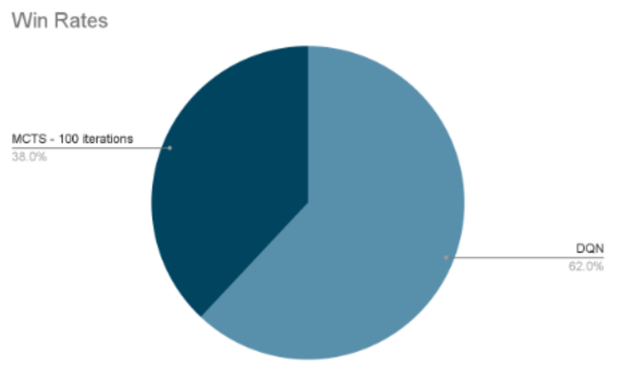
3.4 Deep Q Learning VS MCTS

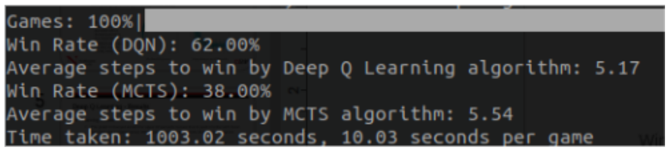
To give the MCTS a fighting chance against the chunky deep learning model, we tried 3 iteration sizes: 10, 100 and 1000. The results ewas monitored for win rates, steps taken and most importantly time taken to complete the game.



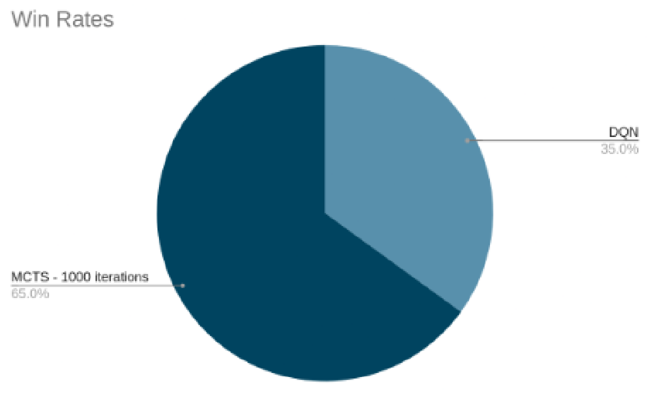


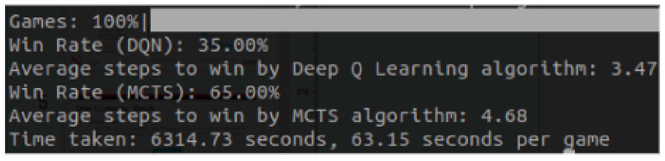
```
Games: 100%|
Win Rate (DQN): 36.00%
Average steps to win by Deep Q Learning algorithm: 5.89
Win Rate (MCTS): 64.00%
Average steps to win by MCTS algorithm: 6.05
Time taken: 107.57 seconds, 1.08 seconds per game
```





```
Games: 100%|
Win Rate (DQN): 62.00%
Average steps to win by Deep Q Learning algorithm: 5.17
Win Rate (MCTS): 38.00%
Average steps to win by MCTS algorithm: 5.54
Time taken: 1003.02 seconds, 10.03 seconds per game
```





```
Games: 100%|
Win Rate (DQN): 35.00%
Average steps to win by Deep Q Learning algorithm: 3.47
Win Rate (MCTS): 65.00%
Average steps to win by MCTS algorithm: 4.68
Time taken: 6314.73 seconds, 63.15 seconds per game
```

4 Conclusions and Comparisons

From the above experiments, we saw that the models performed **equally well** against a **random** opponent, but the details come into play when we put them against each other. Keeping the DQN as the player 1, it was able to beat the MCTS in number of iterations 100. This shows that the model was trained well, but the exploratory advantage of the MCTS helped it beat the DL model when we started randomly.

But the major issue with the MCTS was time taken. From the results above, we see that the MCTS does get better as the iterations increases but the time taken increases exponentially. Almost taking a whole **minute** to complete a game at 1000 iterations per action. This was computationally costly, while the DL model took less than a second per move.

The experiment with MCTS iterations 10, performed really well against the DL model surprisingly, which I would attribute again to the **exploration** done by the tree being built. But this was not consistent when tried again. This is apparent as we saw a **spike in average steps taken to win** the game in the x10 iteration which was higher, showing more randomness than informed actions. Due to time constraints we were not able to test higher iterations multiple times, but they seemed to be consistent enough in 3 tries.

Hence we have an unreliable fast MCTS algorithm that **can** beat the DL model sometimes (x10 iterations), a good reliable MCTS that can beat the DL algorithm but takes forever to make a move (x1000 iterations) and the middle zone (x100 iterations) is dominated by the DL model beating the MCTS starting randomly.

Thereby we conclude by **understanding the tradeoffs** between the algorithms and it just comes down to the compute available and what is more important to us. The DL model is more robust in nature and the MCTS is more efficient if we have enough compute to run it in real time. We also release the code used for the project on my Github: `https://github.com/TarunSrinivas23/Connect4_RL.git`

References

[1] Ankit Choudhary. `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[3] John Tromp. `https://oeis.org/A212693`.