Tic-Tac-Toe and Connect 4 Games with RL

Dharmik Shah
MSc in Computer Science Student
Ryerson University
Toronto, Canada
dharmik.shah@ryerson.ca

***Abstract*—The applications of reinforcement learning have evolved drastically over the decades, ranging from learning in simulated environments to defeating the top human player in the game of Go. This paper explores the applications of reinforcement learning as it is used to learn and play the two-player games of Tic-Tac-Toe and Connect 4. We study four reinforcement learning techniques - QLearning, SARSA, TD0, and finally, the infamous Monte Carlo Tree Search. We compare the impact of learning rates, the number of iterations, the rewarding system, and more to determine which algorithm is best for which game. We discuss our findings in this paper.**

***Index Terms*—reinforcement learning, board games**

I. Introduction and Motivation

Reinforcement learning is an area of machine learning concerned with developing intelligent agents capable of taking actions in an environment to maximize the concept of an expected future reward [1]. It can be thought of as a machine learning paradigm alongside supervised and unsupervised learning. We can prepare the agent in elementary environments such as a 4x4 grid world or very complex ones such as Go. Agents have long been able to self-learn and play games with limited state space, such as a grid world. Within the last decade, research has shown the application of reinforcement learning while playing Atari games such as space invaders through neural networks. In this paper, we will instead train an agent to play the board games of Tic-Tac-Toe and Connect 4. The game of Connect 4 is especially more complex due to the large state size, and we will explore this in-depth throughout the paper [2]. We will use four reinforcement learning techniques - QLearning, SARSA, TD0, along with Monte Carlo Tree Search to train our agent. Our agent will play against itself in these two-player games, learn from these experiences, and in the end, will play against one another to find the best algorithm. We will also analyze the impact of the learning rate, the number of iterations, the rewarding system and more to determine which algorithm is best for which game.

II. Learning Techniques

We have decided to use four learning algorithms to train our agent. The first three: QLearning, TD0 and SARSA, fall under a class of algorithms called Temporal-Difference. The final algorithm: Monte Carlo Tree Search is a heuristic search algorithm used widely in board games. We now provide a brief introduction to how each algorithm works and how the algorithm calculates the next best action.

A. QLearning

QLearning is a model-free reinforcement learning algorithm that learns the value of an action in a particular state, commonly called the state-action value. For any finite MDP, it can find the optimal policy and store it in its $\mathcal{Q}(s, a)$ lookup table. Given a new state, the optimal action to take is the one that maximizes the $\mathcal{Q}$ function for that state [3]. The update rule is as follows:

$$\mathcal{Q}(s_t, a_t) \leftarrow \mathcal{Q}(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a' \in \mathcal{A}} \mathcal{Q}(s_{t+1}, a') \right) - \mathcal{Q}(s_t, a_t)$$

B. TD0

TD0 is the most straightforward TD algorithm that learns the value of a state, commonly called the state-value function. Given a state, we sample all possible moves from this state and take the action that has the max next state $\mathcal{V}$ value [4]. The lookup table $\mathcal{V}(s)$ is updated as follows:

$$\mathcal{V}(s_t) \leftarrow \mathcal{V}(s_t) + \alpha \left([r_t + \gamma \mathcal{V}(s_{t+1})] - \mathcal{V}(s_t) \right)$$

C. SARSA

SARSA is an algorithm that is similar to QLearning, in that it also calculates a $\mathcal{Q}(s, a)$ lookup table. The main difference is in the update step. The SARSA algorithm attempts to explore its options a little more instead of taking the max action all the time [4]. The lookup table $\mathcal{Q}(s, a)$ is updated as follows:

$$\mathcal{Q}(s_t, a_t) \leftarrow \mathcal{Q}(s_t, a_t) + \alpha \left(r_t + \gamma \mathcal{Q}(s_{t+1}, a_{t+1}) - \mathcal{Q}(s_t, a_t) \right)$$

D. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm widely used for solving board games, even those with a very large state space. In the deep learning program called AlphaGo, MCTS is used in conjunction with Neural Networks to defeat the top Go player. MCTS is an algorithm that finds the best action repeatedly by performing four steps: Selection, Expansion, Simulation and Backpropagation [5].

The selection phase involves selecting a node from the ”search tree”. The concept of a node is analogous to a board state. For example, in the game of Tic-Tac-Toe, there are initially nine nodes. These nodes that represent each of the states we can form when we make our first move by placing a marker into any of the nine slots. Each node has an associated score n_s and a field representing the number of times we visit this node, n_v [5]. These are initially both 0. The node with the highest probability of winning is selected first.

This probability is determined in the backpropagation step using the n_s and n_v fields. In the first iteration, all nodes are equally likely to be chosen.

After choosing our node in the selection phase, we can move on to the expansion phase of the algorithm. Here, we randomly sample an additional action from this state. As a result of this, a new state is now born.

In the simulation phase, we randomly play out a game from this new state until we reach a terminal node (hence the Monte Carlo). A terminal state is a winning, losing, or drawing state.

In the backpropagation phase, we inspect the reward of the terminal node from the environment. It will be a reward for winning, losing or drawing the game. The reward is backpropagated to the parent node, which updates its probability values based on the n_s (reward) and n_v (number of times visited). The backpropagation continues on until the root. We determine the probability through the upper confidence bound (UCB1) concept [5]. The probability for a specific node $\mathcal{P}(n)$ is:

$$\frac{n_s}{n_v} + C\sqrt{\ln \frac{np_v}{n_v}}$$

Here, n refers to the current node, and np refers to the parent node. There is a constant C, which is our tuning factor. We have set it to 2.

When the root is updated, another iteration of the tree search begins. We then select the node with the highest probability once again.

A crucial distinction between MCTS and the other algorithms is that MCTS does not build any lookup table. For every new state that forms as the game goes on, the MCTS must rerun its algorithm on this new state [6]. MCTS does not brute force millions of possible states. The key differentiator is that MCTS chooses the best possible action from the tree and only repeatedly searches from that action onwards. In that sense, MCTS visits more interesting nodes more often and thus focuses on the best actions only. The TD algorithms do not have this property [7]. We can now analyze each of the games.

III. Analyzing the Game Tic-Tac-Toe

A. Rules

Tic-Tac-Toe is a relatively simple game that has been around since the time of the Romans. It involves a 3x3 grid in which two players, "X" and "O", alternatively place their respective markers in any empty square. When there is 3 in a row of any specific player marker, that player wins. The sequence can be along the rows, columns, or any diagonal. The game is a draw if no moves are left, and there is no winner.

B. Action Space

The action space for a 3x3 Tic-Tac-Toe is relatively tiny. Initially, there are at max nine actions that player "X" can choose from, corresponding to the nine empty squares. Once they move, player "O" has eight actions left, player "X" has 7, etc. Since the set of actions for any player depends on the number of empty squares, we define the action space as follows:

$$|\mathcal{A}| = 9 - E$$
$$E = |grid[i][j] == EMPTY|, \;\; 1 \leq i, j \leq 3$$

C. State Space

The state-space for a 3x3 Tic-Tac-Toe is much larger than the action space. An overestimate would be 9! since there are nine possible first movies, eight possible second moves, etc. However, this does not consider games that finish in less than nine moves. It can be shown through brute-force or mathematics that the total number of games is 255168 [8].

$$|\mathcal{S}| = 255168$$

D. Learning Techniques

The purpose of analyzing the action and state space is to determine which algorithm is feasible enough to go ahead with, given our time constraints. In Tic-Tac-Toe, the state and action space are small enough to be stored in memory on most modern-day computers. As such, the $\mathcal{Q}(s, a)$ and $\mathcal{V}(s)$ lookup tables for QLearning, SARSA and TD0 are small enough to compute, and so we will use these algorithms. The Monte Carlo Tree Search is also an appropriate learning technique, but we save it for Connect 4.

IV. Analyzing the Game Connect 4

A. Rules

Connect 4 is a 6x7 board game in which two players, "X" and "O", alternate by dropping their chip down any of the columns that have space. The game's goal, similar to Tic-Tac-Toe, is to form a sequence in any direction of the same player. The sequence length must be 4, and whoever achieves this first is declared the winner. The game is said to be a draw if there are no further positions to place chips, and neither player has won thus far.

B. Action Space

The action space for a 6x7 Connect 4 game is small, just like that of Tic-Tac-Toe. Initially, there are at max seven actions that player "X" can choose from, corresponding to the seven columns. Once they move, player "O" can drop their chip down any column that has space. Thus, the set of actions for any player depends on the number of filled columns. We define the action space as follows:

$$|\mathcal{A}| = 7 - C$$
$$C = |\{\text{col } j; grid[i][j] \neq EMPTY, \forall i\}|, 1 \leq i \leq 6, 1 \leq j \leq 7$$

C. State Space

The state-space for a 6x7 Connect 4 game is magnitudes more extensive than that of Tic-Tac-Toe. Initially, there are 42 empty positions, and each block can either be filled by "X", by "O", or none. A wild overestimate is 3^{42} states. However, this estimate does not consider invalid configurations of the board.

For example, an invalid board configuration is when a player places a chip somewhere other than the first available row. The OEIS database has calculated an exact number of states to be 4.53×10^{12} [9].

$$|\mathcal{S}| = 4.53 \times 10^{12}$$

D. Learning Techniques

Due to the ample state space, using any of the lookup table algorithms is not feasible as we could not store these many states in memory. Even if this were possible, calculating these tables would take a very long time. The only feasible solution is to use Monte Carlo Tree Search, which provides an excellent action recommendation despite the longer calculation times. It is much more efficient on memory and time than the other algorithms. We will use Monte Carlo Tree Search as our primary algorithm. We now discuss how the agent is trained.

V. Training the Agent

In a reinforcement learning environment where the agent is the only entity, we can train the agent by repeatedly exploring the environment. An example of this is an agent attempting to learn a grid world. In the case of 2 Player games, we have the flexibility of allowing our agent to play against itself to learn. For instance, let us assume we want to train a QLearning agent to learn either Tic-Tac-Toe or Connect 4. We first create a scenario where two QLearning agents have identical hyperparameters (α, γ). Initially, agent "X" makes a move and updates the $\mathcal{Q}$ table. Agent "O" then views the board, chooses an action using an $\epsilon-$greedy strategy, and updates the same $\mathcal{Q}$ table. Once we reach a terminal state and a winner is declared, the process repeats itself for many games. Having two agents playing against each other helps us determine how good a state-action pair is for both players. In the end, we extract out the $\mathcal{Q}$ table that was common to both players and store it in a file. Using a similar approach, we have SARSA and TD0 agents train amongst themselves. We then extract out the $\mathcal{Q}$ and $\mathcal{V}$ tables and store them in a file.

In each state, our agents will receive a reward $\mathcal{R}(s)$ as follows:

$$\mathcal{R}(s) = \begin{cases} 1 & \text{"X" wins} \\ -1 & \text{"O" wins} \\ 0.5 & \text{draw} \\ 0 & \text{ongoing game} \end{cases}$$

The rewarding system is common for both games. To put the concept of a shared $\mathcal{Q}$ table more concretely, consider the following Tic-Tac-Toe game:

$$\mathcal{Q}\left(\begin{array}{c|c|c} \text{X} & 2 & \text{O} \\ \hline 4 & \text{O} & 6 \\ \hline 7 & 8 & \text{X} \end{array}, 7\right) > 0$$

Notice how since it is the turn of player "X", if they select action seven as their next move, there are guaranteed to win. It does not matter what the following action of player "O" is. In the same $\mathcal{Q}$ table, the following $\mathcal{Q}(s, a)$ would be in favour of player "O" and thus have a negative value:

$$\mathcal{Q}\left(\begin{array}{c|c|c} \text{X} & \text{O} & \text{O} \\ \hline \text{O} & \text{O} & 6 \\ \hline \text{X} & \text{X} & 9 \end{array}, 7\right) < 0$$

As a result, in QLearning, we select the $\max$ action if we are player "X" and $\min$ action if we are player "O". The same logic follows for the remaining agents.

VI. Results

Now that we have discussed how we can train the agents to play a game, we will put it more concretely below:

Algorithm 1: Train QLearning agent

```
QList = []
for α ∈ {0.15, 0.30, ... 0.9} do
    for i = 1 ... 1000 do
        game = Game(QL(α), QL(α), TTT())
        game.play()
    done
    QList.append(game.Q)
done
```

The example above shows how we train a QLearning agent on Tic-Tac-Toe. We consider a variety of α values, and for each α, we initialize two QLearning agents with the hyperparameter. We play 1000 games, in which each agent will update the same $\mathcal{Q}$ table according to their rewards. After all the games finish, we extract the $\mathcal{Q}$ table that indicates the best state-action values for each player. We repeat this process for the remaining α values. In the end, as a result of training our QLearning agents on different values of α, we will now have six uniquely constructed $\mathcal{Q}$ tables.

Once we train QLearning, SARSA and TD0 using the above approach, the next step is to have different agents face off against each other. We will keep the α common and thus decide which agent is better.

Algorithm 2: QLearning vs. SARSA

```
for i = 1 ... 1000 do
    game = Game(QL(ϵ = 0, Q = Q_0.3),
      SARSA(ϵ = 0, QS = QS_0.3), TTT())
    game.play()
done
```

Above, we see a QLearning agent face off against a SARSA agent by playing 1000 games. Notice that we have the hyperparameter $\epsilon = 0$ passed into both agents. The change ensures that the agent picks the optimal move from the lookup table instead of relying on the $\epsilon-$greedy strategy [10]. Also, notice how for both agents, we use $\mathcal{Q}$ table corresponding to $\alpha = 0.3$. As a result, we can analyze the role of α and determine which agent is better.

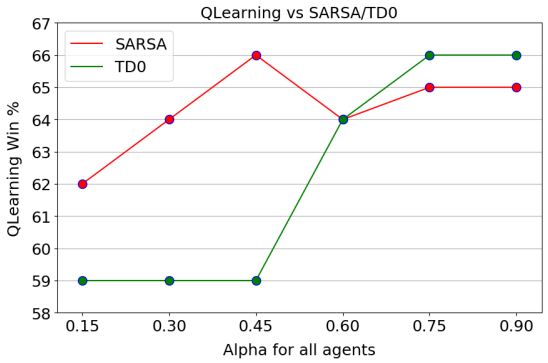
In every move, the agent will view the board state, go through its $\mathcal{Q}$ table, and perform the action that maximizes or minimizes $\mathcal{Q}(s)$. Once all games finish, we calculate the winning % of player one as follows:

$$(\textbf{p1\_wins} + 0.5 \times \textbf{draws})/1000$$

VII. Tic-Tac-Toe Results

We will now discuss our findings by comparing our agents in the game of Tic-Tac-Toe. We train each agent by making it play against itself across 200000 games with various α values. We chose 200000 games because the total amount of Tic-Tac-Toe states is around 255000, so we felt that this was a good stopping point. We will focus primarily on the TD agents, as the Tic-Tac-Toe lookup table is small enough to fit in memory.

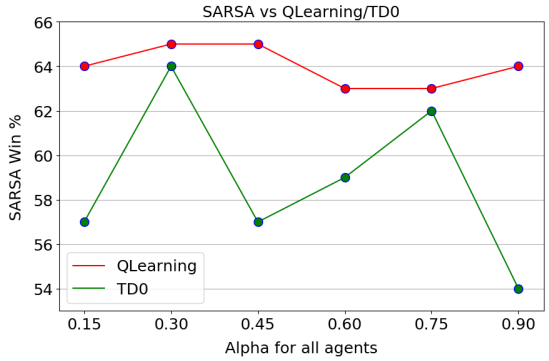
Consider the following graph below, where we compare QLearning against SARSA and TD0, varying the α parameter. Every scatter point represents the result of 1000 games played between QLearning and either SARSA or TD0.



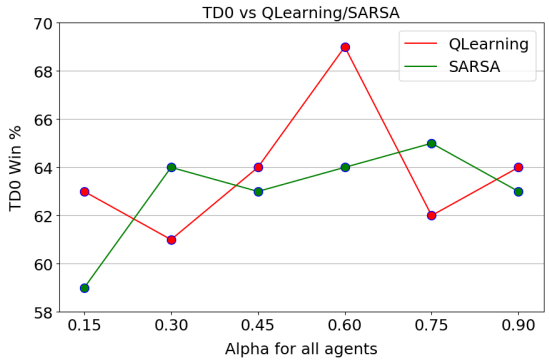
We first notice that when QLearning goes first, it wins around 60 - 70% of its games. We also see that when α is lower, QLearning generally performs better against SARSA than TD. We know that lower values of α mean we learn slower. We also know that SARSA often takes longer to converge compared to TD0. Thus, we expect SARSA to perform poorer for lower values of α [11]. However, we can see that as we increase α, SARSA eventually starts to perform better than TD0, which conforms with our understanding of the two algorithms. Overall, we can see that as α increases, the other two agents perform weaker as they are learning too fast, and this results in a higher win rate for QLearning. QLearning will always choose the max move, and thus it does not seem to be affected by α as much as the other agents are.

We will now discuss how the SARSA agent performed while playing against QLearning and TD0. Consider the graph that follows on the next page.

We first observe that likewise to the QLearning games, SARSA also has a greater than 50% win rate. Perhaps the most striking part of the above graph is that QLearning performed poorer than TD0 against all values of α. We know that QLearning performs better in environments where we do not need to be cautious [12]. In the game of Tic-Tac-Toe, there are very few actions an agent can take.



A single wrong move can result in a victory for the opposite player, thus making it a cautious game. All of these factors justify why QLearning is weaker when playing against SARSA. However, this gives light to an interesting observation. In the last chart, we saw that QLearning performed well against SARSA, but it seems to perform worse in this chart. As a result of these observations, we can conclude that it makes a difference who the first player is in the game of Tic-Tac-Toe. Once the first move is played, it has been shown that player one can always win if the second player doesn't play perfectly.



Shifting gears towards TD0, the win rate against it varies drastically as the α changes. It is not entirely evident what causes this. Still, upon further investigation, it appears that the number of wins for SARSA and TD0 has remained relatively constant, with higher fluctuations in the number of draw games. As a result, the win percentage appears to fluctuate up and down. In reality, the number of games drawn is changing the most.

As with the other charts, it is clear that the first player in Tic-Tac-Toe has a clear advantage, evident from the greater than 50% win rate. We notice that this graph is similar to the first one, in that as α increases, the first agent performs better. Thus, there is an inverse correlation between α and the win rate of player 2. We also see an exciting outlier at α of 0.60, where TD0 won 69% of the games against QLearning and only 64% against SARSA. If we look back at our first graph, we see that for the same α, QLearning has a sharp increase in the win rate against TD0.

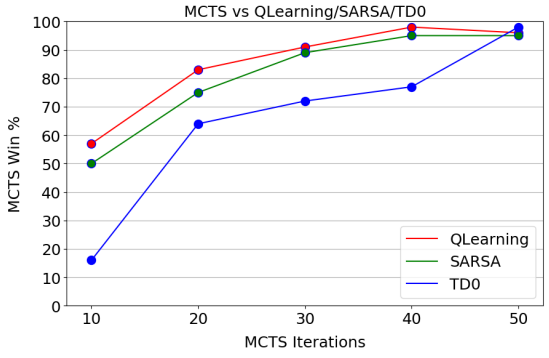
Given that in the newer graph, QLearning is the second player, and we see a similar spike, we can conclude that α of 0.60 is a crucial point for QLearning. Specifically speaking, if QLearning goes first, then at α of 0.60, it performs much better, and if it goes second, it serves much worse. There is likely a deeper cause, but as of now, we have pinpointed a key α value.

Before we end the analysis section for Tic-Tac-Toe, it is worthwhile to discuss the impact of rewards. Our current rewards system provides a reward of 1 for winning the game, -1 for losing, and 0.5 for drawing. When we varied the rewards to be $2, -2, 0$ and $10, -10, 5$, we could not deduce whether or not there was an effect on the agents. Although we did see a change in win rate, it was only evident after α was changed. Thus, we conclude that α is the primary deciding factor. The reward may play some role, but we need to do further testing to demonstrate this.

Overall, we have seen that in Tic-Tac-Toe, the first player has a clear advantage. We have also seen that as α increases, the win rate of the second player goes down the majority of the time.

A. Connect 4 Results

As we mentioned previously, the state space of Connect 4 is too large to be stored in a TD algorithm efficiently. Therefore, we use MCTS as our primary algorithm and TD with a cap limit on the lookup table size. To test our game of Connect 4, we first trained our QLearning, SARSA, and TD agents by having them play 50000 games against themselves. The agent took considerably longer to train itself compared to Tic-Tac-Toe, and thus only α of 0.3 was considered. We then had each TD agent face off against the MCTS agent, which we believed to be the best algorithm for this game. The results are shown below:

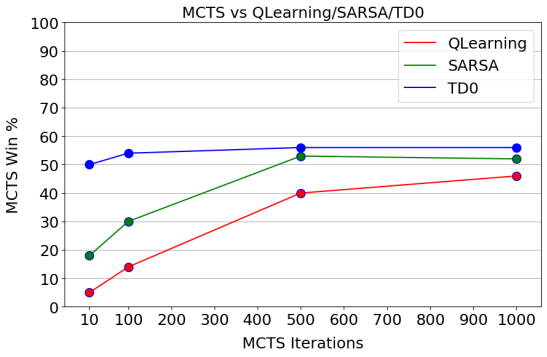


The graph above plots how the MCTS agent performs against the trained TD agents. Instead of varying α, for MCTS, we change the number of iterations. One iteration is defined as when we go through all four phases of the MCTS: Selection, Expansion, Simulation, and Backpropagation. Thus, if we have the number of iterations as 10, this means that we select ten nodes, expand on them, play all ten games out, and then backpropagate the results back to the root. When all iterations finish, we return the node with the highest probability.

Recall that MCTS must be called every time there is a new state, meaning that once player 1 makes a move, we pass in that state to the MCTS. MCTS then tells us the best action to take from there. The algorithm is different from any of the TD ones, where we had a lookup table and could find the optimal move relatively quickly. In MCTS, as the number of iterations increases, so does the time to calculate the best move.

We will now discuss the results of the MCTS games. We purposely had the MCTS agent go second. The game of Connect 4 follows a similar principle to that of Tic-Tac-Toe, in that if player 2 does not play perfectly, player 1 can always be the victor. We train the TD agents to play against the MCTS, where we vary the number of iterations from 10 to 50. We notice that the MCTS performs well for lower iterations, even in a game with a complex state space. Specifically speaking, the MCTS won more than 50% of its games against QLearning and SARSA but had a lot of trouble with TD0 at the start. The likely cause of this is that MCTS, like QLearning and SARSA, takes longer to converge and thus performs poorer on smaller iterations. When the iterations become 20, we see a significant improvement in the win rate over TD0, going from 15% to 62%. As the number of iterations increases, we see that MCTS has a higher win rate each time. By 30 iterations, it is winning more than 70% of all games, and by 50, that number goes as high as 90%. As the number of iterations increases, we can conclude that MCTS will have a higher win rate. One of the main advantages of MCTS is how it searches through the game. The Monte Carlo Tree grows asymmetrically and focuses on the more promising nodes. In the TD algorithms, we are forced to go through most of the game states, and thus games with a high branching factor such as Connect 4 are infeasible for them. MCTS will only expand on nodes with a higher probability and therefore focus on those nodes that are essential to winning the game.

MCTS is an optimal choice in games with large state spaces; however, it is just as good for more minor games like Tic-Tac-Toe. Below, we consider the graph where MCTS is player two and plays against the TD agents in the game of Tic-Tac-Toe. Please keep in mind that the TD agents have been trained in over 200000 games, and this will be the first time MCTS sees this game.



We observe that in the game of Tic-Tac-Toe, ten iterations alone is enough to achieve a 50% win rate against TD0 practically. Initially, MCTS does not do well against QLearning and SARSA, but as the number of iterations increases, MCTS very quickly reaches their level. We notice that by 500 iterations, MCTS has a near 50% win rate against all agents, and by this point, it draws many of the games. As it is player 2, and the TD agents are playing very well as they have been trained on 200000 games, the best thing MCTS can do is draw. We notice that by 1000 iterations, it practically matches their level, and the win rate seems to plateau after that. From this, we can understand that MCTS could reach the TD algorithms level by only playing 1000 games. In contrast, the TD agents needed to play 200000. As it only plays 1000 games, it is relatively quick compared to the TD agents. Thus we can conclude that MCTS is also the optimal choice in Tic-Tac-Toe.

Before we end this section, we would like to say a few words on how the MCTS selects the best nodes to expand on. MCTS uses a concept called the Upper Confidence Bound. The formula is given below, and one exciting hyperparameter to discuss is C. The parameter is essentially the factor that controls the exploration vs exploitation tradeoff.

$$\frac{n_s}{n_v} + C\sqrt{\ln \frac{np_v}{n_v}}$$

The C balances the exploration of unvisited nodes against the exploitation of available rewards. The exploration part is responsible for helping the agent explore and discover new parts of the tree, representing new game states. As a result, the agent can find a more optimal path to the goal. Exploration is used to ensure that MCTS does not overlook potential better paths. The problem with exploration is that it is inefficient for large state spaces, such as Connect 4. To combat this, we have the concept of exploitation. Exploitation enforces that the agent keeps following the path of the largest probability [13]. Exploitation causes the tree's asymmetry and the focus on important nodes rather than all of them. We found that the factor of 2 helped balance the exploration-exploitation tradeoff the best. Other factors considered were 0.5, 1 and 3. We found the best results to be within the range of 1 and 2 and decided to go with 2 for the larger state space of Connect 4.

Overall, in Connect 4, we found that MCTS behaves the best as the number of iterations increases. The C value of 2 resulted in the most wins.

VIII. Implementation and Code

We have implemented all the code in Python using an object-oriented design. A central Game class takes in 2 agents, a board game to play, and a variety of other parameters. The agents (MCTS, QLearning, TD, SARSA) are implemented as classes and can easily be substituted for one another. The board games (BoardTTT, BoardC4) are also classes, and thus with a single line change, the agent learns to play an entirely new game. In addition to the algorithmic agents, there is a Human agent where the user can play against the computer agents. For example, the user can play Connect 4 against MCTS.

If the user decides to play Tic-Tac-Toe, they may face any of the TD agents, which have been trained on 200000 games. The games are also highly configurable. Suppose the user wants to play a 5x5 version of Tic-Tac-Toe instead of a 3x3. In that case, this is done by modifying the corresponding constant. A similar logic holds for the Connect 4 related constants. There exists a demo file and README with more details.

IX. Future Work

One of the main downsides of QLearning is its inability to work well in environments that have larger state spaces. The lookup table in these environments would often not fit in the physical memory. The number of states to calculate would take a very long time. Instead of calculating the Q values through iterating and eventually converging, one working solution is to approximate the Q values using machine learning. More specifically, the Q values can be approximated using a Neural Network. This approach is known as Deep Q Learning, and the neural network is often abbreviated as DQN. The DQN takes as input the game state, and the output is the Q value associated with all possible actions from this state [14]. There are challenges with DQNs, such as representing the game state so the network can understand it, dealing with something we call an "experience replay", and more. However, if trained on enough states, DQN will perform better than Q iteration and will not store any lookup table in memory. DQN will accept a state, and return the best action based on the neural network.

References

[1] LSutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

[2] Ghory, Imran. "Reinforcement learning in board games." Department of Computer Science, University of Bristol, Tech. Rep 105 (2004).

[3] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8.3 (1992): 279-292.

[4] "Reinforcement Learning: Temporal Difference (TD) Learning ..." Reinforcement Learning: Temporal Difference (TD) Learning. Web. 3 Mar. 2022.

[5] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

[6] Chaslot, Guillaume Maurice Jean-Bernard Chaslot. Monte-carlo tree search. Vol. 24. Maastricht University, 2010.

[7] James, Steven, George Konidaris, and Benjamin Rosman. "(PDF) an Analysis of Monte Carlo Tree Search." (PDF) An Analysis of Monte Carlo Tree Search. Web. 3 Mar. 2022.

[8] Bottomley, Henry. "How Many Tic-tac-toe (noughts and Crosses) Games Are Possible?" How Many Tic-Tac-Toe (noughts and Crosses) Games? Web. 3 Mar. 2022.

[9] Tromp, John. "A212693." Number of Legal 7 X 6 Connect-Four Positions after N Plies. OEIS, 23 May 2012. Web. 3 Mar. 2022.

[10] Tokic, Michel, and Günther Palm. "Value-difference based exploration: adaptive control between epsilon-greedy and softmax." Annual conference on artificial intelligence. Springer, Berlin, Heidelberg, 2011.

[11] Sewak, Mohit. "Temporal difference learning, SARSA, and Q-learning." Deep Reinforcement Learning. Springer, Singapore, 2019. 51-63.

[12] Corazza, Marco, and Andrea Sangalli. "Q-Learning and SARSA: a comparison between two intelligent stochastic control approaches for financial trading." University Ca'Foscari of Venice, Dept. of Economics Research Paper Series No 15 (2015).

[13] Coggan, Melanie. "Exploration and exploitation in reinforcement learning." Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University (2004).

[14] Hester, Todd, et al. "Deep q-learning from demonstrations." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 32. No. 1. 2018.