MCTS-AI-playing-Tic-Tac-Toe

Alexander Nihlstrand, Gustav Lundberg

February 29, 2024

1 Implementation

Our implementation utilizes the concept of Monte-Carlo Tree Search (MCTS) which according to Ankit Choudhary on the blog at Analytics Vidhya consists of four major algorithmic parts [1]:

- **Selection** - Selecting good child nodes, starting from the root node R, that represent states leading to better overall outcomes (wins).
- **Expansion** - If L is not a terminal node (i.e. it does not end the game), then - create one or more child nodes and select one (C).
- **Simulation (rollout)** - Run a simulated playout from C until a result is achieved.
- **Backpropagation** - Update the current move sequence with the simulation result.

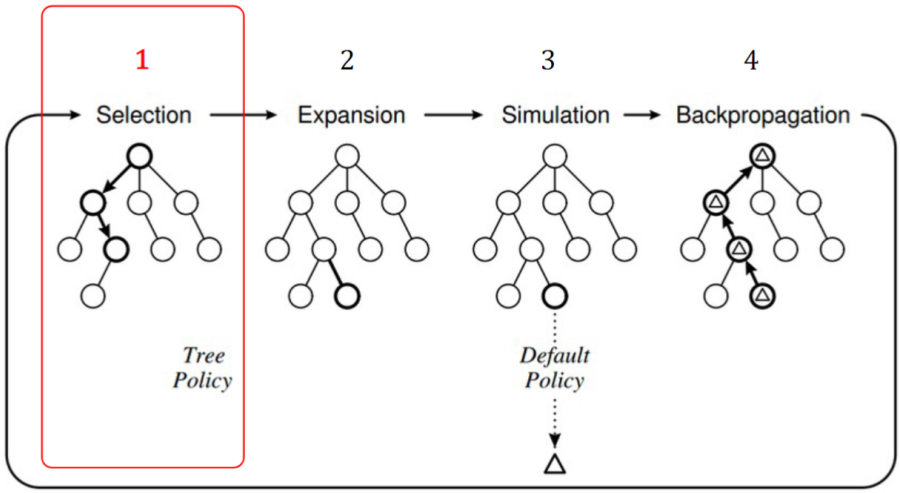

Figure 1: Visualization of MCTS algorithm.

One important distinction from the general MCTS algorithm is that the algorithm knows our opponents policy during the simulation in our implementation.

Player Policy

When it comes to the roll-out policy of our player we use a Monte-Carlo Tree Search to find the optimal moves. The policy iterates through the search tree, which nodes consists of the possible states of a TicTacToe board. Using the node (game-state) we are currently on, we test the possible TicTacToe moves achievable from this current board. Each game state is then assigned a summarized score, which is calculated by simulating possible terminated games from the board state we are currently on after the play. The score of each terminated game is initially +1 for win, 0 for draw and -1 for a loss, which are then summarized for all simulated games into the summarized score.

Once the simulation is done the acquired score of the leaf node from which the simulation started is then back-propagated through the search tree, updating the score values for all parental nodes until the root node. This ensures that the scores for each analysed node are always updated, so that they reflect the best move currently achievable with the information we have obtained this far. However, the algorithm is also limited by a maximum number of iteration. This makes it so that we only receive information from simulating nodes until the maximum iterations are reached, leaving out the results received by simulating more nodes. This results in a trade-off between model performance and number of iterations, essentially translating into time required to run the model.

Opponent Policy

Our opponent policy on the other hand is rather manually written, and created through several checks that mimics the optimal plays of an actual human. The following checks are implemented in this order:

1. Is there any immediate win?
2. Are there any possible blocks?
3. Is the center tile empty?
4. Place marker randomly.

For example, the policy first check if we can instantly win by placing a marker anywhere and does so if possible, and then we do the same check but for possible blocks for the enemy player (block defined as stopping a 2-in-a-row in a 3x3 board). If none of these moves are possible, we check if the center tile is empty and place our marker there, since this is considered a key tile for winning and thus beneficial. If none of the previous options are available, we choose an empty tile randomly.

The randomness in the last choice is interesting, as this gives our player policy a possibility to actually win games. The opponent policy place optimally in almost all cases but can by randomness in the second move of a game make the game lose-able by placing a marker on a side and not a corner if the central tile is taken (mimicking a possible misplay by a human). This aspect makes the games more interesting, as we can identify if our player policy is actually optimal since we can check if it identifies wins when possible instead of only playing for draws.

Search tree selection Policy

The selection policy used when simulating the possible outcomes for our current state consists of a random policy that balances between exploitation and exploration according to our tree policy UCB. We use upper confidence bounds (UCB) in combination with a search tree, creating upper confidence trees (UCT).

Furthermore, this is combined with our opponent that during the simulations acts every other move but using its own policy described above. However, the opponent policy is predetermined for most board states i.e it will always block a 2-in-a-row or win if possible etc. Therefore, the model simulations naturally incorporate sampling, as certain board states will never occur due to the opponent consistently selecting the same move for specific previous boards. This significantly reduces the number of simulations done by the model, making the sampling process efficient, as the MCTS avoids iterating over moves that the opponent will never generate. The balance between exploration and exploitation further enhances the sampling efficiency, as the model's UCT score guides the model to traverse the most promising nodes. Tuning of this balance is later presented.

Reward system

The reward system consisted of updating our visits and wins. Our baseline had the same weights for rewards for wins and losses. A draw made an indifference to the score. These metrics were used for every single test for every grid size. To enhance the model performance these were later tuned.

Evaluation

When assessing the performance for our MCTS player policy without tuning it we get quite good results. When running the model for 100 games we get 35 wins 22 losses and 42 draw. As such the model does manage to draw but it also identifies possible winning moves in an early stage, capitalizing on a misplay made by the opponent that lead into a win several steps later into the game. An example of this intelligence is displayed in Figure 2 below.

Here the algorithm begins by playing on the center tile (X_1) as this is the most beneficial initial move. Then the opponent does a random misplay, placing its marker at the middle-left tile (O_1). The model now already realize that it can win three moves later by playing top-left(X_2) forcing the opponent to block(O_2), and then top-middle (X_3) making the opponent have two rows to block and only being able to block one(O_3). It then wins by playing the tile the opponent doesn't block(X_4).

Winner: X

$$\begin{bmatrix} X_2 & X_3 & O_3 \\ O_1 & X_1 & - \\ - & X_4 & O_2 \end{bmatrix}$$

Figure 2: Example of a game where the model capitalize on a mistake and win the game as X.

This implies that the model to some extent can make intelligent choices and thus performs rather well. However, on the other hand, one may argue that TicTacToe is a rather simple game that the model should never lose, since it is possible to guarantee at least a draw in TicTacToe. As such the model under-performs in this aspect, sometimes missing a crucial move leading to an immediate loss.

However, as mentioned in the implementation paragraph, the maximum iteration criteria was added with a trade-off between performance and run-time. As such we might lose some performance as described above, but in return we gain a lot in the run-time aspect. By making the algorithm search less exhaustively it becomes much faster, while still selecting the best move in most cases.

Taking these aspects into consideration, the model behaves somewhat human-like both by drawing most games but also by capitalizing on mistakes done by the opponent. Additionally, in some cases when not analysing enough due to insufficient number of iterations it might misplay itself, very similar to a human making a overly hasty move overseeing a loosing move. Thereby the model, does lose some matches when matched by a human but if presented with a human misplay it will also utilize this essentially playing rather evenly compared to a human.

4x4 grid & 5x5 grid

Scaling our model into a 4x4 and 5x5 grid worked almost seamlessly as the initial model implementation was rather scaleable. When assessing the performance of the model for the larger grids we got similar results both for the 4x4 grid and the 5x5 grid. When using a proper number of maximum iterations both models behaved significantly better than for lower number of iterations, which is visualized in Table 2 and Table 3. Furthermore, these games resulted in draws for most cases, but the wins were less frequent. As such the game performance was quite similar.

For the run-time, the 4x4 and 5x5 grids are still viable options but the maximum number of iterations becomes even more important to select carefully. While a 3x3 grid gives 9! = 362880 amount of possible games (without accounting for games ending before 9 moves). A 4x4 grid has 16! = 2.092279e+13 possible moves, and 5x5 grid gives 25! = 1.551121e+25.

Since we have a larger grid, the amount of nodes (possible game-states) become significantly larger making each simulation take longer time. To compensate for this, we lowered the maximum number of iterations when increasing the grid size to require less simulations and thus compensating for the vastly increasing number of board states. This ensured that the algorithm was still time-efficient while still performing quite well in most cases, even though we're simulating a smaller share of the total nodes in the search tree due to the iteration roof being lowered. This ultimately leads to somewhat worse performance.

Table 1: Comparison between 1 iteration and 7000 iterations

1 iteration	**7000 iterations**
Winner: DRAW	Winner: DRAW
$\begin{bmatrix} X & O & X \\ O & O & X \\ X & X & O \end{bmatrix}$	$\begin{bmatrix} X & O & O \\ O & X & X \\ X & X & O \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} O & O & X \\ X & O & - \\ X & X & O \end{bmatrix}$	$\begin{bmatrix} X & O & X \\ O & X & X \\ O & X & O \end{bmatrix}$
Winner: O	Winner: X
$\begin{bmatrix} - & O & X \\ X & O & - \\ - & O & X \end{bmatrix}$	$\begin{bmatrix} X & X & O \\ O & X & - \\ - & X & O \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} O & - & - \\ - & O & X \\ X & X & O \end{bmatrix}$	$\begin{bmatrix} X & O & O \\ O & X & X \\ X & X & O \end{bmatrix}$
Winner: O	Winner: X
$\begin{bmatrix} O & O & X \\ O & X & - \\ O & X & X \end{bmatrix}$	$\begin{bmatrix} O & - & O \\ X & X & X \\ X & O & - \end{bmatrix}$

Table 2: Comparison between 1 iteration and 1000 iterations

1 iteration	**1000 iterations**
Winner: O	Winner: DRAW
$\begin{bmatrix} O & X & X & - \\ X & O & O & X \\ O & X & O & - \\ X & X & O & O \end{bmatrix}$	$\begin{bmatrix} O & O & X & X \\ X & O & O & O \\ O & X & X & O \\ O & X & X & X \end{bmatrix}$
Winner: O	Winner: O
$\begin{bmatrix} - & - & O & - \\ X & X & O & X \\ O & X & O & - \\ X & - & O & - \end{bmatrix}$	$\begin{bmatrix} - & - & X & O \\ - & - & X & X \\ - & X & X & - \\ O & O & O & O \end{bmatrix}$
Winner: DRAW	Winner: DRAW
$\begin{bmatrix} X & X & O & X \\ X & O & O & X \\ O & X & O & O \\ O & O & X & X \end{bmatrix}$	$\begin{bmatrix} O & O & X & X \\ X & O & O & O \\ O & X & X & O \\ X & X & O & X \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} X & X & O & O \\ X & O & O & - \\ O & X & O & X \\ X & X & O & - \end{bmatrix}$	$\begin{bmatrix} X & O & O & O \\ O & X & X & X \\ O & X & O & X \\ X & O & O & X \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} O & O & O & X \\ X & O & X & X \\ X & O & X & O \\ X & O & O & X \end{bmatrix}$	$\begin{bmatrix} O & O & O & X \\ X & O & X & X \\ X & O & O & O \\ X & X & O & X \end{bmatrix}$

Table 3: Comparison between Iteration 1 and Iteration 500

Iteration 1	Iteration 500
Winner: O	Winner: DRAW
$\begin{bmatrix} O & X & X & X & O \\ - & O & X & X & O \\ - & O & X & X & - \\ O & O & O & O & O \\ X & X & X & X & O \end{bmatrix}$	$\begin{bmatrix} X & O & O & O & X \\ O & O & X & O & X \\ O & O & X & O & O \\ X & X & O & X & X \\ X & X & X & O & X \end{bmatrix}$
Winner: DRAW	Winner: DRAW
$\begin{bmatrix} X & X & O & O & X \\ O & O & X & X & X \\ O & X & O & O & O \\ X & X & O & X & X \\ X & X & O & O & O \end{bmatrix}$	$\begin{bmatrix} X & O & X & O & O \\ O & O & X & O & X \\ O & O & X & X & O \\ O & X & X & X & X \\ X & O & O & X & X \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} O & O & O & O & O \\ X & O & - & - & O \\ - & - & X & X & - \\ X & X & X & X & O \\ - & X & - & - & - \end{bmatrix}$	$\begin{bmatrix} X & O & O & X & O \\ X & O & X & X & O \\ X & X & O & X & X \\ O & X & O & O & X \\ X & X & O & O & O \end{bmatrix}$
Winner: O	Winner: O
$\begin{bmatrix} - & - & - & - & X \\ O & O & O & O & O \\ - & - & X & - & - \\ X & - & - & - & - \\ - & - & - & X & X \end{bmatrix}$	$\begin{bmatrix} X & O & - & - & X \\ - & O & - & - & X \\ X & O & O & X & X \\ O & O & - & X & - \\ - & O & O & X & - \end{bmatrix}$
Winner: O	Winner: DRAW
$\begin{bmatrix} X & O & X & - & - \\ O & O & O & O & O \\ - & X & X & O & O \\ - & - & X & - & - \\ - & X & - & X & X \end{bmatrix}$	$\begin{bmatrix} X & O & O & O & X \\ O & O & X & O & X \\ O & O & X & O & O \\ X & X & O & X & X \\ X & X & X & O & X \end{bmatrix}$

Tuning reward system

Striking a balance between trying new things (exploration) and using existing knowledge (exploitation) is important for making good moves. Exploitation helps us excel in familiar situations, while exploration allows us to discover new possibilities and avoid getting stuck in routines.

Without the explore component, the model will greedily just traverse through nodes that have high winning rates which ultimately brings few winning/draw playouts. This is especially bad for unpredictable opponents where the model does not have any prior knowledge of the opponents strategies.

Vice versa, having only the explore component, the model will never learn to actually win or tie games properly. This will resemble a "breadth-first search" approach, where it explores every option equally, quickly reaches its computational limits, and ultimately failing to discover winning tactics.

```
exploit = child.total_simulation_reward / child.visits if child.visits > 0 else float('inf')
explore = np.sqrt(2 * np.log(self.visits) / child.visits) if child.visits > 0 else float('inf')
score = exploit + exploration_constant * explore
```

Listing 1: Exploration Term Calculation

The UCT formula in our Monte Carlo Tree Search, as seen in Listing 1, reflects this balance. It considers both the past performance of an option (exploitation) and its potential for discovery based on how often it's been explored (exploration).

The reward system's baseline was adjusted to optimize both exploration and exploitation, aiming to enhance overall performance.

Exploration

In optimizing for exploration we made the model prefer exploring over exploiting in selecting new children. This was done by tuning the the exploration constant. In Table 4 we can see the model treats exploration differently as it affects the total simulation rewards. A higher constant make the model prefer exploration as the exploration becomes larger in relation to exploitation.

Exploration Level	Total Simulation Reward	Exploration Constant
Low Exploration	-18	0.1
Baseline Exploration	-20	1.4
High Exploration,	-69	10
Very High Exploration	-407	100

Table 4: Comparison of Exploration Constants

However, this made us realize that the reward of winning should not be equal to the negative reward of losing. As the model is trained on playing against an opponent with a fairly optimal strategy, the MCTS algorithm will experience more losses than victories. Furthermore, in our baseline model, a draw held no significance, yet a rational player would typically prefer achieving a draw over experiencing a loss during a game characterized by losing playouts.

Exploitation

Based on this we rewrote the whole reward system to more accurately select new children to traverse through in the MCTS algorithm. This was done by tuning exploitation, forcing the model to traverse through nodes with higher winning rates. This was achieved by changing how much the rewards were modified during updates of the backpropagation. The tuning resulted in two different approaches, a defensive approach and an offensive approach.

Table 5: Results of Different Approaches

Approach	Wins	Losses	Draws	Win/Loss Ratio
Baseline	35	22	42	1.52
Defensive	2	10	88	0.2
Offensive	23	13	64	1.77

The defensive approach made the model favour outcomes draws over wins and losses. Here the reward of making a draw was 10 times better than winning or losing. This made the ai win only 2 games, lose 10 games and draw 88 games out of a 100. Whilst decreasing the amount of wins the model also lost fewer games compared to the baseline.

The offensive approach made the model favour wins over losses whilst taking the significance of draws into account. In this scenario, winning was the most rewarding outcome, valued at 25 times the reward for losing and draws were rewarded at twice the value of a loss. This made the ai win 14 games,lose 9 games and draw 77 games out of a 100.

When we compare the win/loss ratio between the defensive and offensive model we can see that the offensive model performs much better, since it does actually outmaneuver its opponent more times than it loses itself thus winning if considering the TicTacToe as a zero sum game. If we instead measure performance through a fewest losses metric we can see that the offensive and defensive strategy performs very similar, both about twice as good as our baseline with 13 and 10 losses compared to the initial 22. Both the offensive and defensive strategy has ”draw” as positive reward, compared to the baseline where ”draw” had 0 as reward. This might explain the considerable reduction in amount of losses for both of the strategies.

Lastly, one more important aspect to consider is that for our implementation the player always begin the game which is a considerable advantage. If the opponent would instead begin half of the games the amount of wins and draws would probably be reduced, but this would influence the performance of all the strategies negatively.

Discussion

Ultimately the model does lose against the opponent in some instances. This outcome can be attributed to various factors. Even if we tried to tune the MCTS, the balancing between exploration and exploitation still face limitations. It still might not explore deeply enough or get stuck on less promising paths, leading

to suboptimal decisions. Additionally, even in simple games like TicTacToe, the vast number of potential moves can overwhelm limited search depths in MCTS. This can prevent thorough exploration of the game tree, potentially missing the best choices.

Furthermore, the MCTS still faces challenges when dealing with dynamic situations. While sampling the opponent's strategies helps account for some unpredictability the alogrithm still has randomness in its simulation where it selects unvisited children nodes randomly. This can lead the algorithm down less favorable paths or miss valuable information, potentially hindering its performance against a skilled opponent who capitalizes on these weaknesses.

This all delves down to time constraints in running a strict amount of iterations for the algorithm. When the MCTS is constrained by time limits, it will not have sufficient time to explore the whole game tree fully to refine its evaluations effectively. The only way to mitigate losses for our opponent policy is if we were to to expand the simulations to the entire game tree with a minimax approach. Even in this case its only possible to draw every single game if both players play optimally. This scenario occurs when neither player makes a mistake and they always block the opponent's winning moves while simultaneously trying to create their own winning opportunities. As our MCTS behaves similiar to how a human would, mistakes are unavoidable, which ultimately makes losses inevitable.

Summing this up we are quite sure that we would manage to beat the ai and have better win percentage against it if we played for infinantely many games since we make less misplays. However as the human error exists (yes, even in us), we would sometime make a misplay which the MCTS could capitalize on and make us lose.

One interesting thing to try different tree policies instead of the UCB policy used in this assignment. For example how a greedy policy would enhance or diminish performance. Since its known to have no exploration this could be a good policy if we have a good approximate of the total simulation reward (Q). But in our case the MCTS knows the policy of the opponent, good estimates of Q could presumably be achieved with some finetuning and thereby mitigating the exploration dilemma. Alternatively, assessing which actions the ϵ-greedy policy would explore, that are known to be bad would be interesting to try out.

References

[1] A. Choudhary, *Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo*, Jan. 2019. (visited on 02/27/2024).