NxN Tic-Tac-Toe with Monte Carlo Tree Search

Abdul Basit Ali

M.S. student in the Khoury College of Computer Science at Northeastern University
ali.abd@northeastern.edu

Abstract

We examine the game of Tic-Tac-Toe on a considerably larger scale for various NxN board sizes. We employ Monte Carlo Tree Search(MCTS)- an online, model-based, reinforcement learning algorithm to select the best moves('X' and 'O'). We then test its effectiveness on multiple sizes of the domain. We have the MCTS agent play against two fixed opponents: a random agent, and itself. The random agent plays valid moves uniformly at random, while the MCTS agent playing itself utilizes its knowledge to play moves for both player 'X' and player 'O'. For games against both fixed agents, we determine the performance of MCTS for each variant domain as a function of hyper-parameters, namely the size of the board and the number of simulations for each planning step. We find that MCTS is exceedingly effective against the random agent as the number of simulations for each planning step increases, winning most of its games. However, we also discover that as the size of the domain(board) increases, MCTS is less effective against the random agent. In addition, we find that MCTS against itself wins less as the number of simulations for each planning step/the size of the domain increases, drawing most of its games. We then compare the effectiveness of MCTS against two powerful temporal difference methods for control: SARSA and Q-learning. We have both methods play against random agents like MCTS. We find that these methods while still effective, are less so when compared to MCTS(with a sufficient number of simulations/planning-step).

Introduction

Tic-Tac-Toe is a turn-based game which has been around for at least hundreds of years, some researchers even dating it back to Ancient Egypt. While the game may seem simple at first within its original domain- a 3x3 board(grid), the complexity starts to increase drastically as the size of the domain increases. This is due to the number of squares composing the board(also known as the state space) increasing. Furthermore, this also leads to the number of valid actions in a given state of the game increasing for both player 'X' and player 'O'. Thus, while the original 3x3 version of the game may seem like a trivial problem to observe and determine win conditions for, it becomes progressively more puzzling and time-consuming for larger versions of the game- especially for humans. Continuing, even many computer-programs are not able to determine sufficient win conditions for large sizes of the domain. This is because techniques employed by such programs can also run into the same problems as humans(large state-spaces, complicated decisions), especially by programs which try to brute-force the best move(s). This is where reinforcement learning can intervene and excel.

Reinforcement learning(RL) is a category of unsupervised machine learning algorithms which aim to learn the optimal policy for a given environment by interacting with it and maximizing a reward signal. The optimal policy can be described as the best way to behave in a given environment. The algorithms are versatile in the sense that they can learn the dynamics(a mapping from states to actions) of the environment just by interacting with it, diminishing the need for constant human intervention(if implemented correctly). It is important to note that the environment can be stochastic, and such control algorithms will still learn the optimal policy in most scenarios.

Reinforcement learning is used frequently in various types of games to find the optimal policy and has proven to exceed human-level performance numerous times. Some relevant turn-based games would be Connect-Four, Chess, and most notably, Go. In March of 2016, the then highest-rated Go player, Lee Sedol, was defeated by AlphaGo- a computer program developed by Google's DeepMind team. AlphaGo utilized many reinforcement learning techniques, one of which being Monte Carlo Tree Search(MCTS). MCTS was primarily used in AlphaGo to select the best moves(actions) in a given state of the game. Using the success of AlphaGo in its win against Lee Sedol and the general success of various RL algorithms in a multitude of games as motivation, we employ and discuss MCTS for another turn-based game: NxN Tic-Tac-Toe.

We aim to show that RL, and more specifically a model-based RL algorithm MCTS, can be used to pick the best moves in a given state of the game, more effectively than other algorithms (Zhang et al, 2022). We show that it does so by a process of planning- thinking ahead, and determining what would happen for a multitude of possibilities, and then choosing the action which maximizes the reward obtained from those possible choices. Furthermore, we discuss the performance of MCTS on various sizes of the domain against two fixed opponents: a random agent and itself.

We show that MCTS is very effective against the random agent(increasingly so as the number of simulations per planning step increases), however, less-so as the size of the domain increases. We also show that MCTS wins less against itself as the number of simulations per planning step/the size of the board increases and why it does so. Finally, we compare MCTS with two prevalent model-free temporal difference methods, namely SARSA and Q-learning. We show that SARSA and Q-learning are also effective at learning the optimal policy, however less-so than MCTS(with a sufficient number of simulations/planning-step).

Background

The Reinforcement Learning Problem

A reinforcement learning problem consists of two main components: the agent and the environment. Starting from an arbitrary state, the agent interacts with the environment by taking an action, and receives some reward and the state it ends up in. It performs this process iteratively, updating its state-to-action mapping which determines how *valuable* it is to be in a state and take a particular action. Formalizing this problem, at time-step t, the agent uses a policy(behavior) function π(s) to determine the appropriate action A_t to take in the current state S_t. Taking action A_t in the environment, results in the agent ending up in a new state S_{t+1} and receiving some reward R_{t+1}.

The data obtained from the process described above, namely (s, a, s', r) is then used to update the policy to learn an improved mapping of state-to-actions. The goal is to achieve an optimal policy defined as π^*, which is the maximum return or reward we can extract from all possible policies. This entire process can be seen in Figure 1.

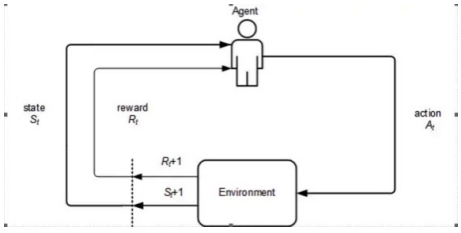

Figure 1: The agent-environment interaction process

Markov Decision Processes(MDPs)

Reinforcement learning is typically viewed as the science of decision-making, as the goal is to learn the optimal policy in a given environment(maximize rewards). Markov Decision Processes(MDPs) describe a model to do just that, providing us with a formal framework for sequential decision making which leads to the policy/policies which maximize the obtainable reward from an environment. Once a problem is modeled as an MDP, it can then be solved for the optimal policy using various RL algorithms.

We can formally describe an MDP as a four-tuple (S, A, R, P). S are the states which make up the state space of the environment, A are the valid actions the agent can take, R is the reward function R(s,a) that represents rewards obtained for all state-action pairs, and $P(s'|s,a)$ are the transitional probabilities of the agent being in the state s, taking action a, and transitioning to a new state s'. For most MDPs, the transitional probabilities and reward function have to be learned by the agent through interacting with the environment. These can then be used to learn the optimal value function $v^*(s)$- the maximal possible return if you start in state s and then follow policy π. We can then use the value function to select the best action in a given state.

Model-Free vs Model-Based Methods

Model-Free RL has no model of the environment as it does not aim to explicitly represent the transition function P or the reward function R. Rather, it focuses on learning the value function through experience by interacting with the environment which allows for optimal action selection.

Model-Based RL consists of the agent learning a model of the environment from experience which allows it to better predict the outcome of taking certain actions (Swazinna et al, 2022). Furthermore, the learned model can then be used to plan(look ahead), which is the process of determining the value function(the best actions to take in the environment). Thus, where model-free RL interacts with the environment directly, model-based RL interacts with the model of the environment through simulated experiences.

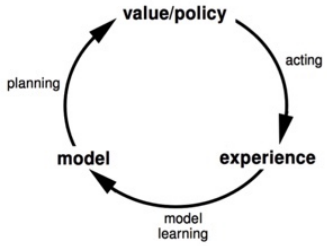

Figure 2: Model-Based RL

SARSA & Q-learning

We now have the required relevant information to discuss two popular reinforcement learning algorithms which learn the optimal policy of an MDP through experience. These methods will be used as baselines against comparisons with MCTS, thus it is vital to understand them. The two methods are State-Action-Reward-State-Action(SARSA) and Q-learning, both of which are model-free temporal difference(TD) methods for control. TD methods aim to estimate expected future returns through the use of bootstrapping. Bootstrapping refers to using the old value as a base to compute the new value. Thus, such methods are useful as they allow the agent to learn from incomplete episodes of experiences, which can help the agent learn more efficiently.

SARSA is an on-policy TD method for control, which means that action selection is done using the policy we are evaluating. Thus, its Q-value updates are done incrementally at every time-step t using the next state s' and the current

policy's action $a^{'}$. We are updating a small amount in the direction of the TD target which is the reward R plus the discounted value of the next state-action pair $\gamma Q(S^{'}, A^{'})$. Figure 3 shows the SARSA update rule.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s'a') - Q(s,a))$$

Figure 3: SARSA update rule

To further explain the update rule, Q(s,a) is the current value estimate for a given state-action pair, α is the step-size parameter, $r + \gamma Q(s^{'}, a^{'})$ is the TD-target described above, and the difference between the TD-target and our current value estimate is defined as the TD error. The goal is to minimize this error in order to learn the optimal value function.

SARSA and Q-learning differ in two ways, both of which can be explained concurrently. SARSA as mentioned previously is on-policy, whereas Q-learning is off-policy. Q-learning is considered off-policy as the Q-value updates are done using the next state and the action is greedy(deterministic with probability 1). Thus, the method in which the Q-values are updated differs from SARSA, and as a by-product, Q-learning is off-policy. The Q-learning update can be seen in Figure 4.

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t,a_t)}_{\text{current value}}\Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Figure 4: Q-learning update rule

To elaborate, Q(s,a) is the current value estimate for a given state-action pair, α is the step-size parameter, r plus the discounted future return is the TD-target, and like SARSA the difference between the TD-target and our current value estimate is the TD error. However, the discounted future return is calculated differently here as mentioned above. It is calculated by iterating over all possible actions a for the next state s_{t+1} and selecting the maximal Q(s_{t+1}, a).

Monte Carlo Tree Search

Monte Carlo Tree Search is a model-based reinforcement learning method. As discussed in the model-free vs model-based section above, model-based methods learn a model of the environment and then use that model to plan. In the case of MCTS, this means that it builds a search tree with the possible actions from a given state, and aims to the find the best action out of the set (Swiechowski et al, 2021). At every iteration it follows the process below, consisting of four primary steps:

- *Selection* - MCTS selects the next node to explore using its epsilon-greedy policy and the current Upper-Confidence-Bound-1(UCB1) values of the nodes. In particular, the maximal UCB1 value of the available action nodes. The UCB1 value helps to balance one of the main problems in RL: exploration and exploitation. While it is important to exploit the actions which yield the maximum reward, it is also essential to explore an epsilon e amount of the time. This helps the algorithm learn about the state-space more deeply and potentially find a better action, generally leading to better results than policies which act only greedily. Figure 5 shows the UCB1

$$\arg\max_a \left\{ \frac{T_a}{N_a} + \sqrt{\frac{2\ln N_s}{N_a}} \right\}$$

Figure 5: Upper Confidence Bound 1 Formula

formula. Here, T_a is the total simulated returns for the proposed action node, N_a is the total number of simulations for the proposed action node, N_s is the number of simulations for the current state-node. The $\sqrt{2}$ is an exploration constant which helps balance exploration and exploitation. Note the *argmax* over all actions a which is used to select the next action based on the node with the maximal UCB1 value.

- *Expansion* - MCTS expands the search tree by adding child nodes for each valid action from the current state, thereby adding new possible states to the tree.
- *Simulation* - MCTS plays out random simulations from the child nodes to terminal states, receiving some reward r. This helps the algorithm determine the result of the action taken, and can be thought of as look-ahead as many *simulated* games are played out from the current state S. This step is essential as it avoids having to brute-force, and thus can explore the important part of the state space efficiently.
- *Backpropagation* - MCTS backs up the values obtained during simulation through the relevant part of the tree. The values can be the number of wins, losses, the average reward, etc. This is crucial as it allows the tree to make better-informed decisions on its next iteration(s), for example, having more accurate UCB1 estimates.

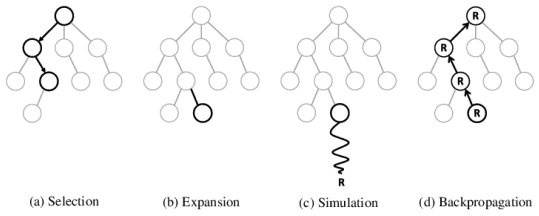

Figure 6: Monte Carlo Tree Search

Related Work

On-Policy Monte Carlo

Monte Carlo(MC) methods like temporal difference methods aim to evaluate and eventually learn the optimal policy. However, unlike TD methods, MC-based algorithms do not use bootstrapping to update their value estimates. Instead, they sample episodic trajectories to terminal states and then

average over the returns to update their $Q(S_t,A_t)$ estimates, where t is the current time-step, S is the current state, and A is the current action. The sampling of trajectories in MC is comparable to the simulation step in Monte Carlo Tree Search(thus the *Monte Carlo* in the name).

While such methods work effectively in many environments, they can be slow in environments with larger state spaces as they require episodic trajectories to calculate returns. Because of this, we utilized TD methods instead as baselines as they can update the policy after each time-step t, thus learning faster.

Dueling Deep Q-Networks

Dueling Deep Q-Network(DDQN) is a powerful variant of a model-free reinforcement learning algorithm Deep Q-Network(DQN). DQN uses a deep neural net to estimate the action-value function. This is similar to the Q-learning update we used, except here a loss function is used. Furthermore, it uses the current value, target value, and reward r to compute the loss. This loss is then used to train the neural net and to improve its estimates.

Dueling DQN separates the representation of its action-value function into state values and an advantage function. The state-value representation as before is the value of being in a given state, while the advantage function is an estimate of taking an action a from a given state s. This split representation of the action-value function helps the agent to learn faster as it can better estimate which actions will produce higher rewards from a given state (Wang et al, 2015). We did not use DDQN as we are trying to determine the effectiveness of Monte Carlo Tree Search, and using DDQN as a baseline would be excessive.

Project Description

Modeling the Environment

We model the environment as an *NxN* grid where $N \in [3,10]$ for the purposes of this project, however the size of the domain is technically unbounded. We then determine the state space to be a grid of the size 3^{N*N}, where the *3* comes from the two actions(*X* and *O)* plus the empty square "", *N* is the number of rows and columns, and *N*N* denotes the total number of squares on the grid. For example, for a 3x3 grid this would mean a state-space of size 3^9. We define the agent stepping through the environment by taking a *valid* action in the form of *(row,col)*, where *row* $\in$ [0,N) and *col* $\in$ [0,N), and *(row,col)* is an *empty* square on the board. We define the terminal states as winning a game, losing a game, and drawing a game. We set the win condition for player *X* or player *O* to be three possible states(resulting as a loss for the other player):

- S_{term} = [($\text{row}_{\text{i} \in [0,\text{N})}$, $\text{col}_{j = 0 \text{ to } j = N-1}$) = ('X' $\oplus$ 'O')] denoting the row win- *N* of the player's symbols(*X* or *O*) in one row, where *N* is the number of rows/columns.
- S_{term} = [($\text{row}_{j = 0 \text{ to } j = N-1}$, $\text{col}_{\text{i} \in [0,\text{N})}$) = ('X' $\oplus$ 'O')] denoting the column win- *N* of the player's symbols(*X* or *O*) in one column, where *N* is the number of rows/-columns.
- S_{term} = [($\text{row}_{i = 0 \text{ to } i = N-1}$, $\text{col}_{\text{j=i}}$) = ('X' $\oplus$ 'O')] denoting the diagonal win- *N* of the player's symbols(*X* or *O*) in one diagonal, where *N* is the number of rows/columns.

We represent a draw state as another terminal state S_{term} = $\neg$([($\text{row}_{\text{i} \in [0,\text{N})}$, $\text{col}_{j = 0 \text{ to } j = N-1}$) = ('X' $\oplus$ 'O')] $\wedge$ [($\text{row}_{j = 0 \text{ to } j = N-1}$, $\text{col}_{\text{i} \in [0,\text{N})}$) = ('X' $\oplus$ 'O')] $\wedge$ [($\text{row}_{i = 0 \text{ to } i = N-1}$, $\text{col}_{\text{j=i}}$) = ('X' $\oplus$ 'O')]) $\wedge$ [($\text{row}_{i = 0 \text{ to } i = N-1}$, $\text{col}_{j = 0 \text{ to } j = N-1}$) = ('X' $\vee$ 'O')] denoting a draw- the absence of a win, and the state-space being fully occupied with no empty squares.

Upon reaching a terminal state, the agent receives a reward. We define the reward function for MCTS/SARSA and Q-learning games for either player *X* or *O* as:

$$R = \begin{cases} 1/100 & \text{if } S_{term} = win \\ -1/-100 & \text{if } S_{term} = loss \\ 0 & \text{if } S_{term} = draw \\ 0/-1 & \text{otherwise} \end{cases}$$

We decide to emphasize the reward for SARSA and Q-learning games and we add a -1 reward for every time-step t to allow the agent to learn faster. We set the draw reward to zero for both MCTS and SARSA/Q-learning games as we would like the agent to prioritize winning.

Implementing Monte Carlo Tree Search

We implement Monte Carlo Tree Search as a set of nodes, each node storing a unique state S of the game board(environment). These nodes store unique properties such as if the state is a terminal state S_{term}, a set of children nodes(states) denoted as $c = \{S_{c1}, S_{c2}, S_{c3} \ldots\}$, a parent node(state) represented as S_p, total returns T_a, and total visits N_a.

We then use the state-nodes to build a search tree from a specified starting-state $S_{initial}$, for a specified number of simulations. Each simulation consists of:

- Selecting the maximal node S_{max} if the current node S_{curr} is expanded fully. We do so by comparing the UCB1 value of S_{curr} with its children nodes c and returning one of the maximal nodes S_{max} at random.
- If S_{curr} is not fully expanded we expand the node for all valid states $S_{valid} = \{S_1, S_2, S_3 \ldots\}$ from S_{curr}.
- We then perform simulation from the expanded node, randomly generating valid states from S_{valid} until we encounter a terminal state S_{term}, receiving some reward $r \in 0, 1, -1$.
- We finally backpropagate up the search tree updating the values for each node S_{curr} until we reach $S_{initial}$: $N_a = N_a + 1$, $T_a = T_a + r_{obtained}$.

Implementing SARSA and Q-learning

We implement SARSA and Q-learning similarly, with the two differences being the update rule and Q-learning being off-policy. First, we implement the policy π as a function of the Q table, the current state S, and an epsilon value e. We employ epsilon-greedy, where with 1-e probability we select the maximal action a_{max} from all valid actions $A_{valid} = \{A_1, A_2, A_3 \ldots\}$, for the state S, which maximizes $Q(S, A_n)$. Then, with probability e we select a random action from A_{valid}.

For both SARSA and Q-learning we initialize a Q-table denoted as Q with zeroes. We initialize a table called F to keep track of the number of wins w, number of losses l, and the number of draws d for the final hundred games(episodes). We then loop for a specified number of episodes *ep* and for each episode we initialize the state s using the environment *env*, we receive the action a using s from the policy we defined above. Then, we *loop* until a terminal state S_{term} is reached, at every time-step t we step through the environment taking action a, we obtain the next state $s^{'}$, the reward r, if this is a terminal state S_{term}, and the current status of the game $g \in (win, loss, draw, in-progress)$. Here is where SARSA and Q-learning differ. For SARSA, we derive the next action $a^{'}$ using $s^{'}$ from the policy we defined above. We then update the Q-table using SARSA's update rule: $Q(s, a) = Q(s, a) + \alpha * (r + \gamma * Q(s^{'}, a^{'}) - Q(s, a))$. We can then set s and a to $s^{'}$ and $a^{'}$ respectively. For Q-learning we loop over all valid actions A_{valid} for $s^{'}$ and select the maximal Q-value Q_{max}. We then use Q_{max} to update $Q(s, a)$ using the following update rule: $Q(s, a) = Q(s, a) + \alpha * (r + \gamma * Q_{max} - Q(s, a))$. We set s to $s^{'}$. Continuing, for both SARSA and Q-learning we check if S_{term} is true and *break* out of the loop if so. Finally, after each episode, if the current episode $ep_{curr} + 100 >= ep$, we update the status of F. We do this to keep track of the outcomes of the final 100 games.

Experiments & Results

The Process Behind the Experiments

For Monte Carlo Tree Search, we decided to test the agent against two fixed opponents for each variant domain(3x3-10x10), as a function of the the number of simulations for each planning step. The motivation behind this was to examine the impact on performance of the size of the domain and the number of simulations for each planning step. The first opponent we had MCTS play was itself. We wanted to observe how the agent would learn the *best* moves for both player X and O, and if it would win less as a result of it. Continuing, we had MCTS play against a random agent which would select valid moves on the board uniformly at random. We performed this experiment to test how well MCTS would perform against a player of *regular* strength.

As baselines for comparisons against MCTS, we used SARSA and Q-learning. Both agents played against random agents like MCTS. The motivation behind using these TD methods as baselines was to determine if a model-based method like MCTS would perform better than model-free methods like SARSA and Q-learning, and if so, then to what degree.

The Monte Carlo Tree Search experiments against both the random agent and itself were ran for number of simulation steps *steps* = 25, 50, 100 and for board sizes *sizes* = [3x3, 10x10]. We decided to use 25 as the starting point for *steps* as we noticed that for a value less than 25, the agent would make sub-optimal decisions. Furthermore, we capped the size of the domain at 10x10 as the time to play out a sufficient number of games was significantly higher as the size of the board and the number of *steps* increased. The SARSA and Q-learning experiments against the random agent were also ran for board sizes = [3x3,10x10], using the hyper-parameters $p = (\alpha : 0.5, \gamma : 0.9, \epsilon : 0.1, episodes : 1000)$. We used an α value of 0.5 to keep the weight of the last-update at a mid-point between [0,1], and we discovered that it generally improved the performance of the algorithms the most in comparison to other step-size parameters. Continuing, we used a γ value of 0.9 as we wanted to be able to learn decently fast while still giving appropriate weight to future rewards. We used an ϵ value of 0.1 to balance exploration with exploitation. Finally, we ran the tests for 1000 episodes as we discovered that the methods generally converged around the 800 episode mark.

Results

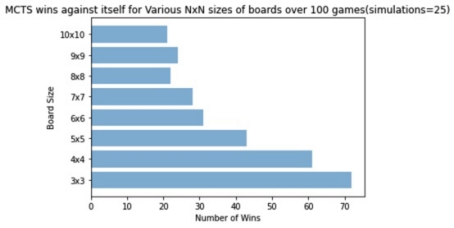

Figure 7: MCTS vs Itself[simulations/planning-step=25]

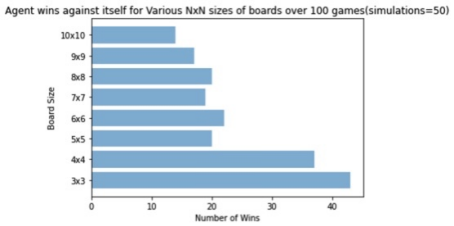

Figure 8: MCTS vs Itself[simulations/planning-step=50]

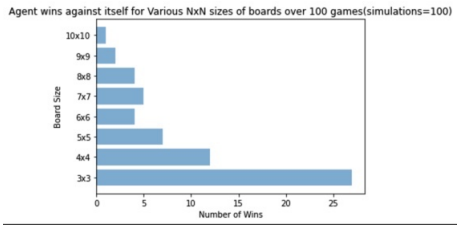

Figure 9: MCTS vs Itself[simulations/planning-step=100]

Figures 7-9 show the results of MCTS playing vs Itself for board sizes = [3x3, 10x10] and for simulations per planning step = (25,50,100). General trends we can observe are that as the number of *steps* increases, the agent wins less. This is because it knows more about the state space and thus can choose better actions. These actions include *blocking* actions- actions where the agent tries to stop itself from winning in a given state of the game S, thus ending a majority of the games in draws. Furthermore, as the size of the board increases, the number of wins also decrease. This is

most likely because it becomes significantly more challenging for the agent to make *optimal* moves, as it has a much larger state space to explore.

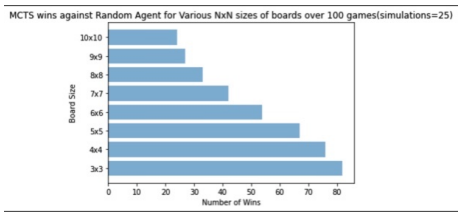

Figure 10: MCTS vs Random Agent[simulations/planning-step=25]

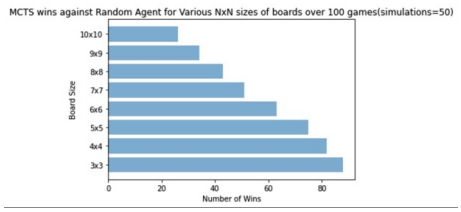

Figure 11: MCTS vs Random Agent[simulations/planning-step=50]

Figures 10-12 show the results of MCTS playing vs the Random Agent for board sizes = [3x3, 10x10] and for simulations per planning step = (25,50,100). General trends we can observe are that as the number of *steps* increases, the agent wins more. This is because it knows more about the state space as its search tree is substantially more dense, and thus can choose better actions against the random-agent who is oblivious to the dynamics of the environment. Furthermore, as the size of the board increases, the number of wins also decrease. This is most likely similar to the problem with MCTS playing Itself, it becomes significantly more challenging for the agent to make *optimal* moves, as it has a much larger state space to explore, thus even against the random agent, it wins less.

Figures 13-14 show the results of Q-learning and SARSA playing vs the Random Agent for board sizes = [3x3, 10x10] and for a 1000 episodes per board. While both SARSA and Q-learning win less than MCTS vs the Random Agent for the first couple of board sizes, both the algorithms seem to perform equal or better than MCTS(*steps*=100) vs the Random Agent for the rest of the board sizes. Furthermore, with SARSA and Q-learning the number of wins for each board-size are much more consistent, with Q-learning maintaining a win-percentage of greater than 60% for all board-sizes. This is most likely due to the nature of TD-methods being effective while exploring a relatively small state space thanks to bootstrapping, thus being able to maintain a decent win-rate for all boards. However, we predict that with more simulations per planning-step, MCTS would outperform both TD-methods as it would have a much deeper understanding of the state-space, where-as both SARSA and Q-learning seem to converge for the hyper-parameters used

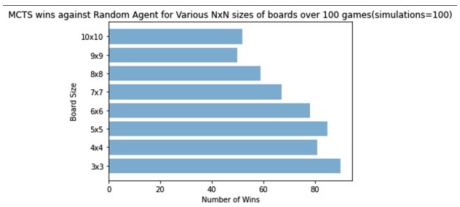

Figure 12: MCTS vs Random Agent[simulations/planning-step=100]

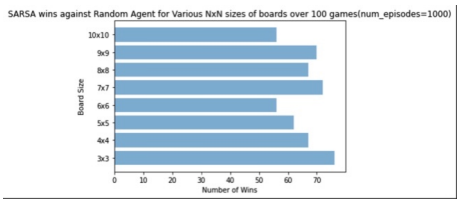

Figure 13: SARSA vs Random Agent[episodes=1000]

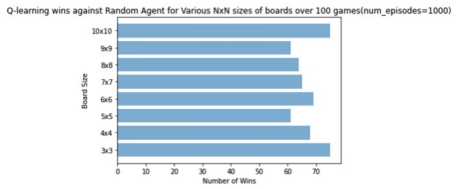

Figure 14: Q-learning vs Random Agent[episodes=1000]

in these experiments. This is where model-based methods such as MCTS excel as they are able to plan using the model of the environment, and thus have a more in-depth understanding of the state-space.

Conclusion

We learned that Monte Carlo Tree Search is a very effective model-based RL method which excels in turn-based games such as NxN Tic-Tac-Toe. Furthermore, we discovered that as the number of simulations per planning-step increases, MCTS wins more against the random agent and less-so against Itself(due to better *blocking* moves). Continuing, we found that as the size of the domain(board) increases, the number of wins for MCTS against both fixed opponents decreases. Finally, upon comparing MCTS to two model-free methods SARSA and Q-learning, we found that they perform as effectively as MCTS for most board-sizes. However, we also discussed that SARSA and Q-learning seem to have converged, and thus MCTS would likely perform better with more simulations per planning-step as it would be able to explore the state-space extensively.

References

Comparing Model-free and Model-based Algorithms for Offline Reinforcement Learning Swazinna, P., Udluft, S., Hein, D., Runkler, T. 2022. arXiv. https://doi.org/10.48550/arXiv.2201.05433

Monte Carlo Tree Search: A Review of Recent Modifications and Applications Świechowski, M., Godlewski, K., Sawicki, B., Mańdziuk, J. 2021. arXiv. https://doi.org/10.1007/s10462-022-10228-y

Dueling Network Architectures for Deep Reinforcement Learning Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N. 2015. arXiv. https://doi.org/10.48550/arXiv.1511.06581

An Efficient Dynamic Sampling Policy For Monte Carlo Tree Search Zhang, G., Peng, Y., Xu, Y. 2022. arXiv. https://doi.org/10.48550/arXiv.2204.12043

Link to Code