Reinforcement Learning

Georgios Kyziridis[1] and Michail Tsiaousis[1]

Leiden University, LIACS, The Netherlands
s2077981@umail.leidenuniv.nl
s2082896@umail.leidenuniv.nl

Abstract. Learning algorithms such as Monte Carlo Tree Search (MCTS) and Q–Learning comprise the backbone of state–of–the–art algorithms, such as Deep Q–Network (DQN) [6] and AlphaGo [7]. Evaluation and comparison of these core algorithms is essential to their theoretical analysis and integration to more advanced ones. This paper evaluates and compares Monte Carlo Search (MCS), MCTS, and Q–Learning at two board games: TicTacToe and Connect4. We find that MCS outperforms the other algorithms with respect to win–rate, convergence time, stability (variance), and sensitivity to different board sizes. MCTS performs fairly well in specific game configurations, but its win–rate is consistently worse than MCS. Q–Learning has, on average, the lowest win–rate, and its performance decreases significantly for large board sizes.

Keywords: Reinforcement Learning · Monte Carlo Tree Search (MCTS) · Monte Carlo Search (MCS) · Q–Learning

1 Introduction

Reinforcement Learning is the field that studies goal–directed learning through trial–and–error interactions with an environment [10,5]. In recent years, there has been an increasing interest in the field of Reinforcement Learning, thanks to state–of–the–art results in game playing, such as Atari [6], Go [7,9], Chess, and Shogi [8], among others. The algorithms responsible for the aforementioned results use as backbone learning algorithms such as MCTS and Q–Learning. Therefore, evaluating and comparing such core algorithms is important in order to understand their theoretical properties, improve them, and integrate them into more complex algorithms. Board games provide the opportunity to evaluate and compare such core algorithms. In this paper, we implement, evaluate, and compare MCS, MCTS, and Q–Learning at two 2–player zero sum board games: TicTacToe and Connect4.

Our contributions are as follows: We examine the 1) convergence, 2) sensitivity to hyper–parameters, games, and board configurations, 3) variance/stability with respect to different conditions (board size, random seeds) of the three algorithms, and finally, 4) the change in performance of Q–Learning when trained against different opponents.

The paper is structured as follows. In section 2, we define the game environments used in this paper. Section 3 presents basic terminology and definitions

of Reinforcement Learning used throughout the paper. In section 4, we present the approaches and algorithms in detail. Section 5 provides the experimentation setup and the research questions we aim to answer. Section 6 presents the results and section 7 concludes the paper.

2 Environments

In this paper, we implement three agents namely, MCS, MCTS, and Q–Learning, to compete against an opponent player on two board games, TicTacToe and Connect4. The environment implementations of the two games are available on Github[1]. The TicTacToe environment is defined by its size m, which produces an $m \times m$ board. The Connect4 environment requires a tuple of (height$\times$width, win_length). A state s of the environment represents the current board in the game, and the available actions from s are all the valid actions from that board. Finally, we define a high–level function called, the Arena. The latter provides the agent (or the opponent) the current state of the environment (board), in which, the agent (or the opponent) has to play an action.

We experiment with two opponents. The agents pit against a random player at TicTacToe. At Connect4, they play against a random player, and a OneStepLookAhead player. The latter is a simple player who always takes a win if presented, or blocks a loss if obvious, otherwise is random. Finally, we use three configurations for each game: TicTacToe 3×3, 4×4, 5×5, and Connect4 (4×5, 3), (5×6, 4), and (6×7, 4).

3 Preliminaries

The environment is represented as a Markov Decision Process (MDP), and in this paper, we limit our scope to discrete, finite MDPs. In this framework, the environment is modeled in discrete time steps $t = 0, 1, 2 \ldots$. At time step t, the agent is situated at state $s_t \in \mathcal{S}$ and takes an action $a_t \in \mathcal{A}(s)$ from that state so as to move to the next state s_{t+1}. Upon exiting the state, the agent receives a numerical signal called the reward, $R_{t+1} \in \mathcal{R} \subseteq \mathbb{R}$. In finite MDPs, the sets $\mathcal{S}, \mathcal{A}, \mathcal{R}$ have a finite number of elements. As the agent continues to interact with its environment, a trajectory or history of state–action–reward triplets is formed: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$. The total cumulative reward is referred to as the return. The goal of the agent is to select actions that maximize the expected discounted return, G_t,

$$\mathbb{E}[G_t] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{1}$$

where $0 \leq \gamma \leq 1$ is the discount rate. The discount rate γ controls the way future rewards are weighted. For $\gamma = 0$, the agent maximizes only immediate

[1] https://github.com/suragnair/alpha-zero-general

rewards, whereas greater values of γ also take into account future rewards. Maximizing only immediate rewards is not an optimal strategy, which brings us to the dilemma of exploration and exploitation. An agent that cares only about immediate rewards will not explore parts of the state space that will possibly generate a lot of reward in the future. On the other hand, in late stages of learning, exploitation ensures that optimal actions are chosen. Consequently, the agent needs to balance exploration and exploitation so as to converge to an optimal strategy of how to behave in the environment.

The agent requires a mapping from states to actions to act in the environment. A discrete policy, π, is a function that maps a state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}(s)$, that is, $\pi(s) = a$. A stochastic policy, $\pi(a|s)$, is the probability of taking action a in state s. The optimal way of behaving in the environment, i.e maximizing G_t, is defined by an optimal policy π^*. Optimal decisions require a notion of how good it is to be in a state, or, how good it is to take an action from a state. The value of state s under a policy π, $v_\pi(s)$, is the expected return from state s, following π onward,

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{ for all s} \in \mathcal{S} \quad (2)$$

The value of taking action a in state s under a policy π, $q_\pi(s, a)$, is the expected return starting from state s, taking action a, and following π onward,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \quad (3)$$

An optimal policy π^* corresponds to an optimal state–value function v^*, and an optimal action–value function q^*. Having either v^* (in model–based methods) or q^* (in model–free methods) is equivalent to know the optimal policy π^*, and thus, the optimal way to behave in the environment.

4 Approaches & Algorithms

4.1 Monte Carlo Search

Monte Carlo Search (MCS), also known as Plain MCS, Pure MCS, or Monte Carlo Evaluation [1], is a rollout–based algorithm used in two–player zero–sum games. MCS calculates, for a given state, the expected outcome of a game given random play (from that state) until the end of the game [1]. In the framework of reinforcement learning, given a state s, MCS estimates $q_\pi(s, a)$ for all possible actions a, assuming that the agent follows a random policy π from state s onward. Policy π is also called a rollout policy. The opponent follows a random policy as well.

In more detail, for a given state s (e.g given from arena), the agent has to choose the best action to play (e.g in arena). For state s, an action is selected

and a complete game is simulated, until termination. This is done for all possible actions from s. This process is repeated until a specified number of iterations is reached, or, until a time constraint prohibits further simulations. $q(s, a)$ is estimated as the average return across the number of simulations, $N(s, a)$, executed from state–action pair (s, a). For state s, the agent chooses as best move the action $a = \text{argmax}_a\, q(s, a)$. Algorithm 1 in Appendix illustrates the pseudocode of MCS.

4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an algorithm for selecting the optimal decision from a given state, by building a search tree based on random samples of the decision space [3]. MCTS has been used in many board games, such as chess, shogi [8], hex [2], and played an important role in the success of AlphaGo [7], the computer program that beat one of the best human player at Go.

MCTS is a rollout algorithm at its core, with the addition of building a tree based on statistics of previous Monte Carlo simulations so as to estimate more accurately the values of possible actions from a state [10,4,12]. The algorithm consists of four steps: selection (traversal), expansion, simulation, and update. In *selection*, starting from the root node, the tree is traversed following a tree policy until a leaf node is reached, that is, a node of which its children have not been added to the tree. In *expansion*, the children of the aforementioned node are added in the tree. A child node is selected based on the tree policy and a *simulation* (rollout) is carried out following a rollout policy until the end of the episode. The return generated by the simulation is used to *update* the average value of every node (state) in the trajectory of the tree policy (from root node to child node). The visit count of every node in the trajectory is updated as well. As a tree policy, we use the Upper Confidence Bound (UCB), which balances exploration and exploitation. The child node j of the parent node p is chosen, that maximizes

$$UCB(s_j) = v_j + C\sqrt{\frac{\ln n_p}{n_j}} \tag{4}$$

where v_j is the average value of child j, n_p, and n_j are the counts of the parent and child node, respectively. $C \geq 0$ is a constant for the trade–off between exploration and exploitation. Values closer to 0 indicate more exploitation, whereas greater values favor exploratory moves throughout the tree traversal.The rollout policy is a random policy over the possible actions from a given state, as in the case of MCS.

For a given state s (e.g given from Arena), the algorithm is executed, that is, a tree is built having state s as root node, for a specified number of iterations, or, until time runs out as in MCS. The chosen action (e.g in Arena) is the one that maximizes the expected return from state s onward.

4.2.1 Implementation Details Our MCTS implementation uses recursion to traverse the tree, see pseudocode of algorithm 2 in Appendix. It should be noted that in two player games, the opponent is also modeled inside the MCTS algorithm. In this paper, the opponent follows the same tree policy as our agent, maximizing eq. (4). The states in the trajectory corresponding to the player that executed the rollout simulation are updated with the correct sign (multiplied by 1) of the generated return. The states of the other player are updated with the opposite sign (multiplied by -1). In this way, the agent learns how to differentiate between actions that lead to high versus low expected return, i.e good versus bad actions.

4.3 Q–Learning

This section refers to the Q–Learning algorithm which was a breakthrough in the reinforcement learning field at the early 90's. Q–Learning is an off–policy algorithm, which means that it uses the (exploratory) behaviour policy to learn the target policy, π, in contrast to on–policy methods, where the agent updates only one policy gradually through time. Q–Learning uses the ϵ–greedy algorithm as behaviour policy, and a greedy (action that maximizes Q–value) as a target policy.

The Q–Learning algorithm is part of a general algorithm called Temporal–Difference (TD), which is used for control. The learned action–value function Q approximates directly q^* according to [10], assuming that all state–action pairs continue to be updated. The following equation defines the Q–Learning update rule based on the Bellman equation in TD control.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \underbrace{\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]}_{TD-Residual} \tag{5}$$

The above equation depicts the update rule of the Q–Learning agent using the TD residual. It updates gradually the $Q(S_t, A_t)$ value. It can be observed that the current Q–value for state s_t, is updated by the sum of a proportion α (the learning rate) of the reward, R_{t+1}, plus the discounted maximum Q–value of the next state s_{t+1}, minus the current Q–value, $Q(S_t, A_t)$.

4.3.1 ϵ–greedy The ϵ–greedy algorithm defines the behavior–policy, where the agent chooses an action completely at random with probability ϵ, or the action with the maximum Q–value with probability $1 - \epsilon$. The referent algorithm handles the trade–off between exploration and exploitation inside the behavior–policy, by controlling the parameter of randomness. Let $\epsilon \in [0, 1]$ be the probability that the agent selects an action completely at random. The following equation describes the binary decision between random and non–random action.

$$\pi(s_t) = \begin{cases} \text{random action from } \mathcal{A}(s_t), & \text{if } \xi < \epsilon \\ \text{argmax}_{a_t \in \mathcal{A}(s_t)}\ \mathcal{Q}(s_t, a_t), & \text{otherwise} \end{cases} \tag{6}$$

where $\xi \in [0, 1]$ is a uniform random number drawn at each time step, t. Ideally, exploration is preferable in the early stages when the agent has limited experience. After collecting experience, the agent exploits its knowledge and selects actions greedily.

4.3.2 Implementation Details There are many variants of the Q–Learning algorithm. The classic algorithm described above is suitable for single–player environments. In our case (board–games), the algorithm has to involve the opponent player. We chose to model the opponent as part of the environment. Thoroughly, given a state state s_t, we consider the next state s_{t+1} as the state after both players have made their moves. Therefore, in state s_t, the agent takes action a_t and the opponent takes an action which leads the agent to the next state, s_{t+1}. The above process is repeated from state s_{t+1} to s_{t+2}, until the final state s_T. The update takes place after the opponent's move, or whenever the game is terminated.

Regarding ϵ–greedy, ϵ decreases using a discount factor $\delta \in [0, 1]$, each time the agent selects a random action. In this way, we slightly decrease the chance of a random action selection according to our requirements. After a finite number of training–episodes, the agent uses the trained Q–table in order to compete in a "real" environment (in Arena), with an opponent, following the target policy (greedy). The training process is described in algorithm 3 in Appendix.

5 Experiments

5.1 Research Questions - Experimentation setup

The following list summarizes the research questions we want to answer and evaluate with our experiments.

RQ_1**:** Compare the methods according to time convergence of the win–rate
RQ_2**:** How hyper–parameter tuning impacts the convergence
RQ_3**:** Inspect the convergence stability on different conditions
RQ_4**:** Trade–off between the time convergence and the stability
RQ_5**:** Examine win–rate sensitivity on different game configurations
RQ_6**:** Performance of the Q–agent trained against skillful opponents

All the experiments were made in the DataScience-Lab of LIACS using a Python framework for parallel programming called JobLib. The machine we used (adamant) has 16 cores (32 threads): Intel Xeon E5-2630v3 CPUs @ 2.4Hz, and 512 GB of RAM, running CentOS 7. The following software and modules were used for the experiments: `Python 3.6.0`, `Numpy 1.13.3`, `Pandas 0.23.4`, `Joblib 0.11`, `tqdm 4.31.1`.

We monitor the performance (win–rate) of MCS and MCTS playing 100 games in Arena, for different response time–values, meaning that the algorithms are executed until time runs out. More precisely, given a state from Arena, the algorithm has a fix response time to run in order to select the action to be played

in Arena. The following response time–values in microseconds are used: 10, 50, 100, 250, 500, 750, 1000, 1500, 2000, 3000 ($\times 10^3$). A single simulation consists of 100 games in Arena for a specific response–time. We run 10 simulations in parallel, mapping each one to a core.

For Q–Learning, we examine the win–rate convergence in both training and Arena, for different training episodes. Since Q–Learning is an offline algorithm, the agent has to be trained before competing in Arena. During training, we extract the last updated Q–Table on specific training episodes, and the agent competes against the opponent in Arena (using this fix Q–Table). The win–rate in training and Arena can be thought of as evaluating the agent on "training" and "test" data, respectively. For all experiments concerning the Arena, the agent plays 10 repetitions of 100 games, and the average win–rate is calculated. For TicTacToe 3×3 and Connect4 (4×5, 3), the agent is trained for a total of 300×10^3 episodes, and the win–rate is calculated every 10×10^3 episode steps. The respective total episodes and episode steps for TicTacToe 4×4 and Connect4 (5×6, 4) are 500×10^3 and 20×10^3, while for 5×5 and (6×7, 4) we use 700×10^3 total training episodes and 28×10^3 episode steps.

5.2 Experimentation Details

The MCS experimentation was made by running simulations for different seeds. More precisely, we ran 5 different simulations (5 random seeds) for each time–value, for each game configuration. In this way we can draw conclusions about the convergence and stability (variance) of win–rate under different conditions (random seeds, game configurations).

MCTS includes the parameter of exploration, C, which has a strong impact on the results in Arena. We first ran simulations in order to explore the performance of the agent under different values of C. We experiment with 6 different values ($C = 0, 1, 2, 3, 4, 5$), for each game configuration, and we monitor the performance of the agent in different time–values. As optimal C, we consider the one with the maximum average win–rate, averaged across the time–values. Afterwards, for each game configuration, we use the optimal C parameter to ran simulations for 5 random seeds, as in MCS. In this way, we can compare the convergence, the overall performance, and the win–rate variance of the two algorithms.

For Q–Learning, we experiment with different values of learning rate α (0.01, 0.05, 0.1), and with two types of ϵ–greedy policies: fix $\epsilon = 0.2$ and dynamic ϵ, where in the first case, following the protocol in [11], exploration stops at $\frac{2}{3}$ of the total training episodes. In the latter case, ϵ starts at 1 and dynamically decreases each time a random action is chosen, with a discount factor of 0.99. Exploration stops when $\epsilon < 0.001$. Our aim is to examine the win–rate convergence with respect to different combinations of α and ϵ, and identify the most sensitive parameter, across different game configurations. For all experiments, we use discount factor $\gamma = 0.99$. Finally, a Q–learning agent is trained from scratch against a more competent player, the OneStepLookAhead player. Our goal is to

examine whether the performance increases when the agent competes in Arena against a random, and a OneStepLookAhead player.

6 Results

6.1 MCTS - Selection of Optimal C

The current subsection describes the MCTS–Agent results for tuning parameter C, over different time–values. Due to space limitations, we present only the most interesting plots for each game–opponent combination, as shown in fig. 1. The plots visualize six different curves, one for each value of parameter, C. The x–axis indicates the response time of the agent for playing an action in Arena, and the y–axis corresponds to the win–rate.

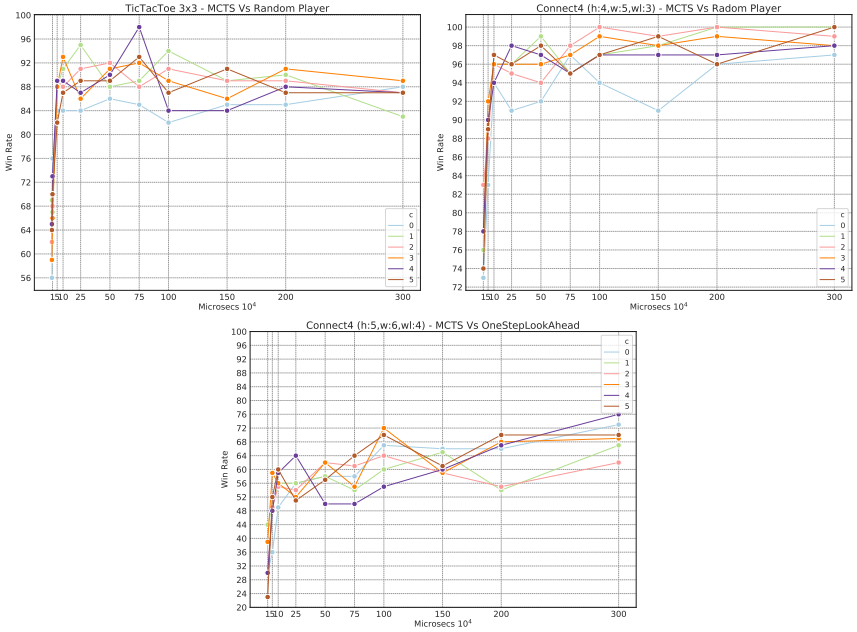

Fig. 1: MCTS win–rate for each exploration parameter C, at TicTacToe 3×3, Connect4 (4×5, 3), Connect4 (OneStepLookAhead) (5×6, 4).

On the left–hand side of fig. 1 (TicTacToe 3×3), the agent starts from 56% win–rate for $C = 0$ (no exploration at all) at 10^4 microseconds, while it reaches the highest win–rate for $C = 4$ at 75×10^4. The variance over the different curves indicates that the agent is sensitive to the exploration parameter C.

On the right–hand side of fig. 1, it is clear that the performance of the agent at Connect4 (4×5, 3) is better, on average, than TicTacToe 3×3. More precisely,

the curves for different C have a stable tendency, fluctuating between 94 to 98, except for $C = 0$. The curve for $C = 5$ fluctuates across all time–values, which indicates that it has not converged. Reasonably, the agent performs better in this game due to the limited state–action space. An interesting fact is, that there are cases where the agent has a 100% win–rate. On average, the curve for $C = 0$ has the lowest win–rate which is reasonable due to scant exploration.

In the final plot, the agent does not perform well against the OneStepLookAhead player. There is big variance between the points of the different curves, as well. Although the curves depict a lower win–rate on average, the MCTS–Agent wins the opponent most of the times. The curve for $C = 4$ reaches the maximum win–rate at 300×10^4, and it seems that it has not converged yet. It is obvious that the exploration parameter, C, has a strong impact on the performance and convergence of the algorithm.

The optimal C for all the experiments for different games and configurations are illustrated in table 1. The games played against the random player resulted in small exploration values as optimal ones, while bigger values of C are optimal against OneStepLookAhead. Intuitively we can conclude that the more skillful opponent, the bigger exploration values are needed.

	TicTacToe			Connect4 (RP)			Connect4 (OP)		
Game config	3×3	4×4	5×5	4×5	5×6	6×7	4×5	5×6	6×7
C	1	1	2	2	1	1	4	3	5
Avg. win–rate	85.72	73.63	54.09	95.20	94.50	93.90	81.70	59.10	65.50

Table 1: Average win–rate of MCTS for best parameter C. RP and OP refer to random and OneStepLookAhead player, respectively.

6.2 MCS & MCTS

6.2.1 TicTacToe Figure 2 includes two different plots. The plots illustrate the average win–rate of MCS and MCTS over 5 different seeds. The standard deviation over seeds is indicated by vertical bars. In this way, we can draw a conclusion about the stability of the algorithms on different repetitions.

MCS performs better than MCTS in all TicTacToe configurations. As the board size increases, the win–rate of MCTS decreases, while MCS's win–rate is insensitive to the different board sizes. In the smallest board, both agents have an immediate sharp increase on the win–rate. Generally, MCS has lower variance over the different seeds, except for TicTacToe 3×3. On 3×3, shown in left-hand side of fig. 2, it is not clear whether MCS has converged. On the contrary, MCTS converges from 100×10^4 at 88% win–rate. On 4×4 (right-hand side of fig. 2), both agents converge at 100×10^4; MCS at 94 and MCTS at 82, with a slight increase for the last microseconds for MCTS. In the big board, fig. 11 in Appendix, the agents converge slower, where MCS seems to converge from 100×10^4 around 92, while MCTS converges from 150×10^4 at 66.

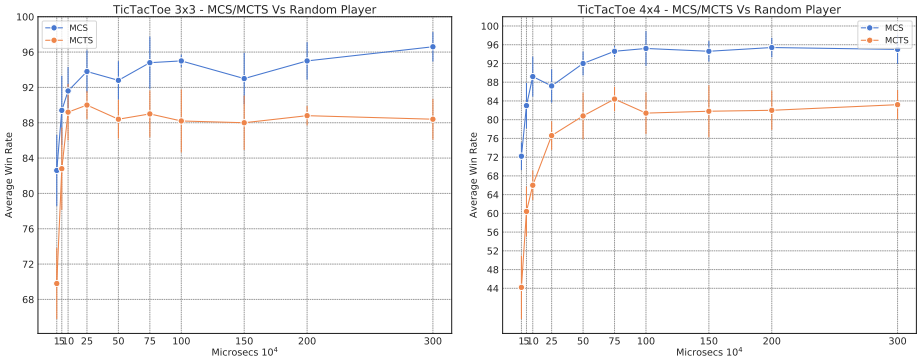

Fig. 2: MCS and MCTS average win–rates for 5 different seeds, at TicTacToe 3×3, 4×4

6.2.2 Connect4 Vs Random Player Figure 3 illustrates the win–rate of MCS and MCTS at Connect4 against the random player. Both agents perform well for all configurations. On (4×5, 3), there is significant overlapping between the two agents' curves except from 150×10^4 microseconds onward, where MCTS is better than MCS. Overall, both agents' win–rates, after stabilizing in the plateau, fluctuate between 97-100. Hence, the agents are capable of beating the random player in all three Connect4 configurations, and they produce quite robust and stable performance. On (5×6, 4), shown on the right-hand side of fig. 3, MCS converges immediately for 50×10^4 at 100, while MCTS converges slower and smoother for 100×10^4 at 97. In fig. 12 of the Appendix, MCS converges slightly faster than MCTS from 75×10^4 at 100 win–rate, while MCTS has a win–rate of 99. Clearly, MCS has lower standard deviation over the 5 seeds in all Connect4 configurations.

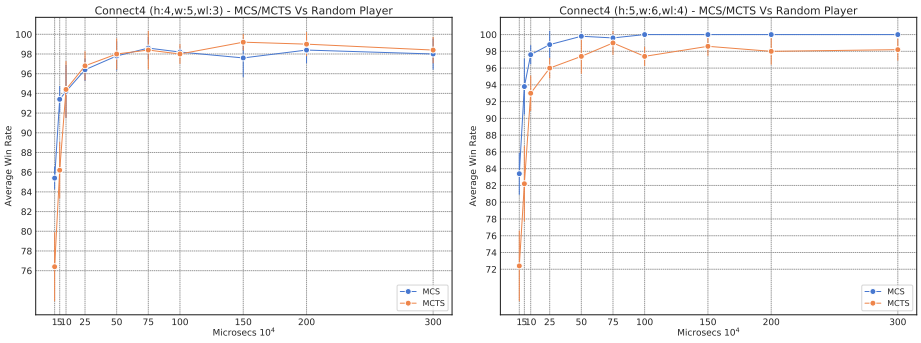

Fig. 3: MCS amd MCTS average win–rates for 5 different seeds, at Connect4 (4×5, 3), (5×6, 4).

6.2.3 Connect4 Vs OneStepLookAhead Figure 4 illustrates the win–rate curves of MCS and MCTS versus OneStepLookAhead, at Connect4. In this case, against a more skillful opponent, it can be observed that the two agents have significant differences regarding their performance. MCS performs better than MCTS with respect to win–rate, and it has lower variance over the 5 different seeds. Overall, MCS converges faster than MCTS in all configurations. In the medium board (right–hand side), MCTS performs worse than in the large board in fig. 13 of the Appendix, while MCS seems to be insensitive to the board changes.

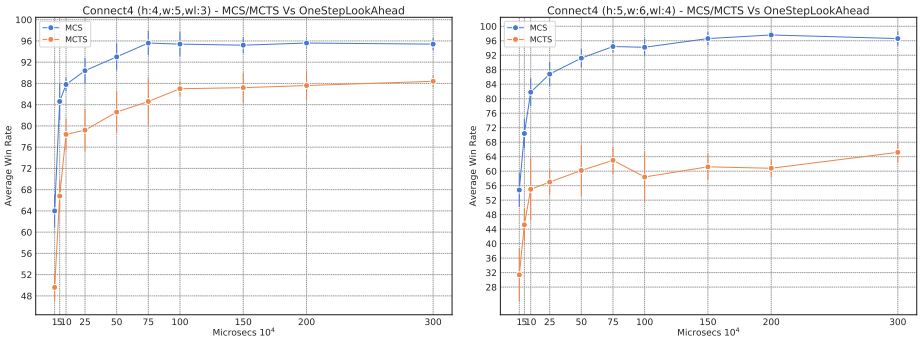

Fig. 4: MCS and MCTS average win–rates against OneStepLookAhead for 5 different seeds, at Connect4 (4×5, 3), (5×6, 4).

6.3 Q–Learning

6.3.1 TicTacToe The win–rate of Q–Agent in the training procedure against a random player at TicTacToe 3×3 and 4×4 is shown on the left-hand side of fig. 5 and fig. 6, respectively. At 3×3, on 10×10^3 episodes, the win–rate ranges already between 0.73-0.8, depending on the combination of α and ϵ. Therefore, the agent is capable of producing fairly good results on 3×3 in few training episodes. On the other hand, on 4×4 the win–rate at 20×10^3 episodes is considerably lower, at 0.3. In both cases, there is a large increase in win–rate for fix ϵ due to the stop of exploration. That is, the agent acts greedily with respect to the action-value function from $\frac{2}{3}$ of the total training episodes onward. This increase lead fix ϵ to surpass dynamic ϵ in both configurations, for all learning rates, with $\alpha = 0.1$ having the highest win–rate of 0.96 and 0.83, respectively. On the larger 5×5 configuration depicted on the left-hand side of fig. 14 (Appendix), the agent fails to learn a good strategy against a random player. All combinations of ϵ and α produce a win–rate of approximately 0.2, across all training episodes.

The right-hand side of fig. 5 and fig. 6 depict the average–win rate in Arena at TicTacToe 3×3 and 4×4, respectively. We observe that the results are in

agreement with those of the training procedure, that is, fix ϵ with $\alpha = 0.1$ is the best combination in both training and Arena for both configurations. Additionally, this combination converges to an average win–rate close to those obtained in training, i.e 0.955 and 0.82 for 3×3 and 4×4, respectively. Training and Arena results agree for the 5×5 board, as well; the average win–rate fluctuates around 0.2.

As expected, the win rate in training and Arena decreases as we move from smaller sizes to larger ones, since the state–action space increases. For the same reason, the training win–rate is more sensitive to the choice of ϵ.

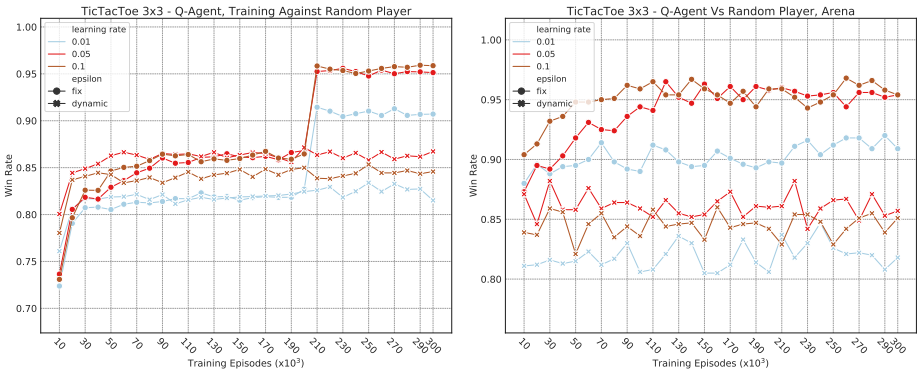

Fig. 5: TicTacToe 3×3. Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

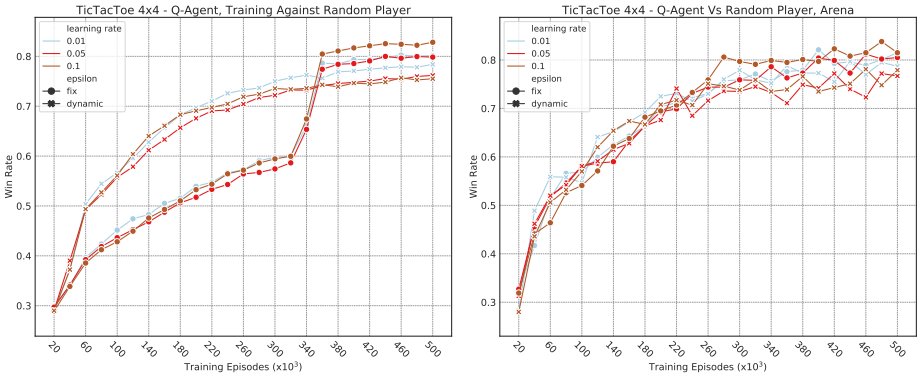

Fig. 6: TicTacToe 4×4. Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

6.3.2 Connect4 The respective plots for training at Connect4 (4×5, 3), (5×6, 4), (6×7, 4) are shown on the left-hand side of fig. 7, fig. 8, and fig. 15 (Appendix), respectively. In all cases, fix ϵ surpasses dynamic shortly after exploration stops, for all learning rates. In all three configurations, the win–rate is fairly high already from the start of training, indicating that the agent is able to perform fairly well without extensive training. On (4×5, 3) and (5×6, 4), the best combination consists of fix ϵ with $\alpha = 0.05$, with a win–rate of 0.98 and 0.93, respectively, at the end of training. On (6×7, 4), fix ϵ with $\alpha = 0.1$ performs slightly better compared to $\alpha = 0.05$, reaching 0.94 win–rate.

The right-hand side of fig. 7, fig. 8, and fig. 15 (Appendix) illustrates the average win–rate in Arena at Connect4 (4×5, 3), (5×6, 4), (6×7, 4). In general, fix ϵ exhibits slightly better performance than dynamic ϵ. For all configurations, all learning rates (for fix ϵ) exhibit similar results. For fix ϵ on (4×5, 3) and (5×6, 4), the best combination in training (fix ϵ, $\alpha = 0.05$) displays an average win–rate of 0.71 at the end of training, in both cases. On (6×7, 4), the best combination found in training (fix ϵ, $\alpha = 0.1$) has a win–rate of 0.695 at the the total number of training episodes. For all configurations, the average win–rate in Arena displays a considerable decrease of approximately 20-30% (depending on the configuration) in comparison to training. This result validates our intuition that it is important to monitor the win–rate in both training and Arena. This decrease does not take place in TicTacToe, which suggests that the agent is able to generalize better on test data at the latter board game. It is interesting to note that in the training procedure, the agent for dynamic ϵ on (6×7, 4) is insensitive to α.

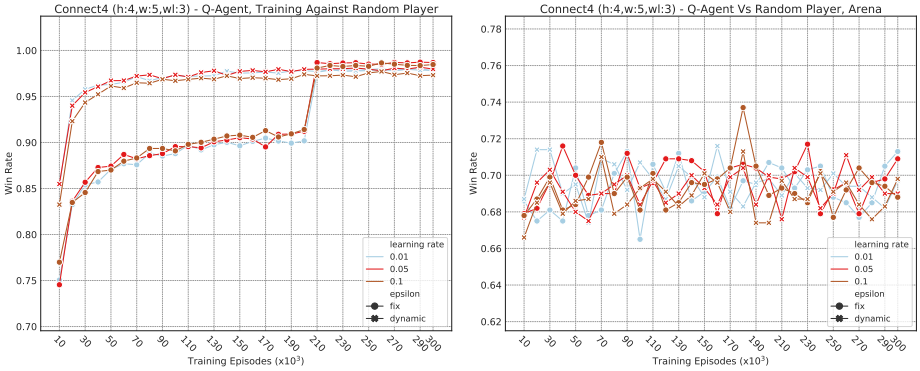

Fig. 7: Connect4 (4×5, 3). Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

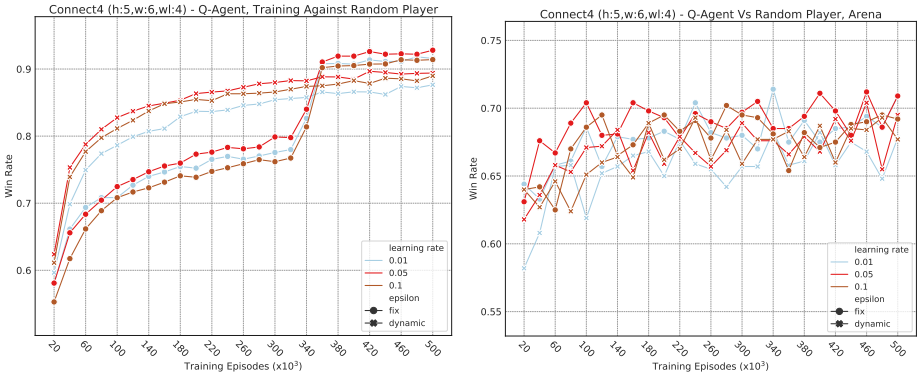

Fig. 8: Connect4 (5×6, 4). Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

6.3.3 Q–Agent against OneStepLookAhead Figure 9 illustrates the average win–rate in Arena of the Q–Agent vs OneStepLookAhead player, when the former was trained against a random player. For all configurations, fix ϵ performs better than dynamic. On (4×5, 3) and (6×7, 4), $\alpha = 0.1$ has the highest average win–rate of 58.5 and 32, respectively, at the of end training. Accordingly, this is a decrease of 11% and 32% for the two board configurations against a random player (see right-hand side of fig. 7 and fig. 15). On (6×7, 4), dynamic ϵ performs even more poorly, managing just above 10%. On (5×6, 4), $\alpha = 0.05$ exhibits the best performance, converging to 0.27 at the end training, which translates to a decrease of 44%, see right hand side of fig. 8.

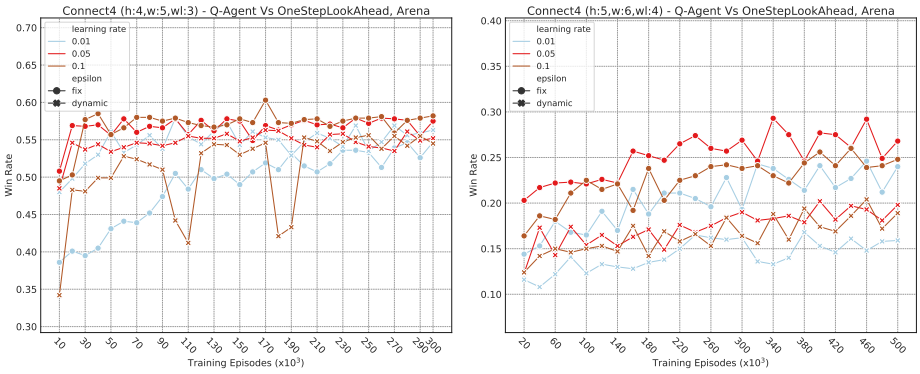

Fig. 9: Win–rate of Q–Agent against OneStepLookAhead in Arena at Connect4 (4×5, 3) (left) and (5×6, 4) (right).

Figure 10 illustrates the win–rate performance of Q–Agent against a random player and an OneStepLookAhead player in Arena at Connect4 (4×5, 3) and (5×6, 4), in which, the agent is trained against a OneStepLookAhead player. We use fix $\epsilon = 0.2$ and $\alpha = 0.01$. The OneStepLookAhead player is more competent in comparison to the random player, so we trained the agent for more episodes, $200{\times}10^4$ and $400{\times}10^4$ for (4×5, 3) and (5×6, 4), respectively. Our aim is to investigate whether the performance of the agent increases when trained against a more competent player.

By comparing the blue curves of the left and right-hand side of fig. 10 with the light blue curves for fix ϵ on the right-hand side of fig. 7 and fig. 8, we conclude: Training against the OneStepLookAhead player does not provide any improvements when the agent competes in Arena against the random player on neither (4×5, 3) nor (5×6, 4). Similarly, by comparing the orange curve on the left-hand side of (4×5, 3) and the light blue curve (fix ϵ) on the left-hand side of fig. 9, we find that training against the OneStepLookAhead player does not provide any improvements even when the agent competes in Arena against the same player. Nevertheless, we observe an improvement of 7-10% on (5×6, 4) when playing in Arena against OneStepLookAhead by comparing the orange curve on the right-hand side of fig. 10 with the light blue curve (fix ϵ) on the right-hand side of fig. 9. Although training takes place for different number of episodes (millions vs thousands), it seems that the light blue curve has converged. Hence, the aforementioned 7-10% increase is due to training the agent against a more competent player, and not because of extensive training.

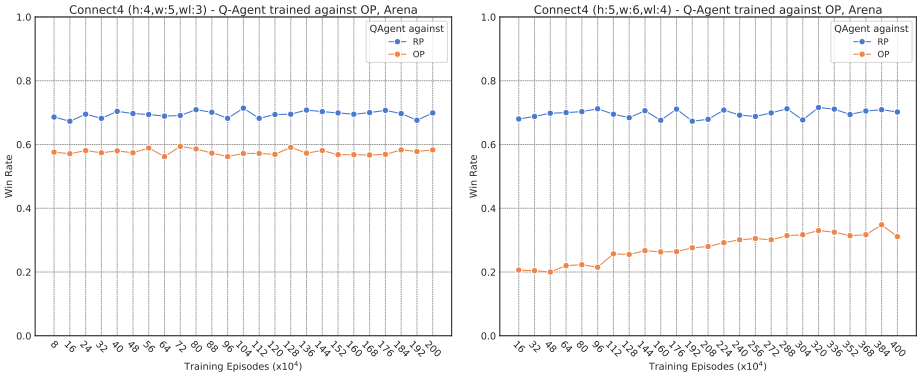

Fig. 10: Win–rate of Q–Agent against the random and OneStepLookAhead player at Connect4 (4×5, 3) (left) and (5×6, 4) (right). The agent is trained against OneStepLookAhead.

7 Conclusion

The current paper examines the performance of three different learning algorithms, MCS, MCTS and Q–Learning, playing two different board games namely, TicTacToe and Connect4. We conclude that, MCS, in comparison to the rest of the algorithms, exhibits the best performance with respect to win–rate, which does not decrease considerably across different games and configurations, that is, the algorithm is insensitive to different games and board sizes. MCS converges faster than MCTS at Connect4, while no conclusion can be drawn for TicTacToe. Additionally, MCS has lower variance (more stable) than MCTS with respect to random seeds and game configurations. MCTS performs fairly well in specific game configurations, but its win–rate is consistently worse than MCS. It is clear that MCTS requires more time to grow the tree enough in order to accurately estimate the action values of the current state. On the other hand, Q–Learning has the slowest convergence, since it requires considerably more training time to reach the performance of MCS and MCTS. Additionally, Q–Learning exhibits, on average, worse win–rates in comparison to the other agents, i.e. it fails to learn a competitive strategy against the random player at TicTacToe 5×5. Furthermore, it is the most sensitive algorithm with respect to different game configurations.

Hyper–parameter tuning significantly affects the performance and convergence for MCTS and Q–Learning. We find that the exploration parameter C affects the convergence considerably. Moreover, smaller values of C are optimal when playing against the random player, whereas larger values are needed when competing against the competent, OneStepLookAhead player. For Q–Learning, fix ϵ performs better than dynamic. The choice of ϵ, affects the win–rate greater than the different values of α. For certain configurations of Connect4, the Q–Agent can be improved when the latter is trained against the OneStepLookAhead and competes with the same opponent in Arena, at the cost of extensive training time (millions versus thousands of episodes). Finally, we observed a large decrease in win–rate from training to Arena at Connect4, which suggests that it is important to monitor both the training and Arena win–rates.

References

1. Abramson, B.: Expected-outcome: A general model of static evaluation. IEEE transactions on pattern analysis and machine intelligence **12**(2), 182–193 (1990)
2. Arneson, B., Hayward, R.B., Henderson, P.: Monte carlo tree search in hex. IEEE Transactions on Computational Intelligence and AI in Games **2**(4), 251–258 (2010)
3. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
4. Chaslot, G.M.B., Winands, M.H., van Den Herik, H.J.: Parallel monte-carlo tree search. In: International Conference on Computers and Games. pp. 60–71. Springer (2008)

5. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of artificial intelligence research **4**, 237–285 (1996)
6. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
7. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484 (2016)
8. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
9. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
10. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
11. Wang, H., Emmerich, M., Plaat, A.: Monte carlo q-learning for general game playing. arXiv preprint arXiv:1802.05944 (2018)
12. Winands, M.H., Björnsson, Y., Saito, J.T.: Monte-carlo tree search solver. In: International Conference on Computers and Games. pp. 25–36. Springer (2008)

A Pseudocodes

Algorithm 1 MCS

Input: Board = $s \in \mathcal{S}$
Output: Action
Initialize $Q(s, a)$ for all $a \in \mathcal{A}(s)$ with zeros
init_board = Board
for simulation = $1, 2, \ldots, N$ **do**
    valid_actions = GetValidActions(init_board)
    for action in valid_actions **do**
        Board = init_board
        Board = GetNextState(Board, action)
        while game is not ended **do**
            a = RandomAction(Board)
            Board = GetNextState(Board, a)
        end while
        R = GetReward(Board)
        $Q(s, a) \leftarrow Q(s, a) + R$
    end for
end for
Calculate the expected return for each state action pair (s, a)
$Q(s, a) = Q(s, a)/N$
Action = $\text{argmax}_a \, Q(s, a)$

Algorithm 2 MCTS part1

```
procedure SIMULATE(Board, cur_player)
    Input: Board = s ∈ S, Current Player
    Append state in trajectory
    if board is not in state_memory then
        Append board in state_memory
        if game is ended then
            UPDATE(board)
        end if
        Q = ROLLOUT(board, cur_player)
        UPDATE(board)
    end if
    if game is ended then
        exit()
    end if
    EXPAND(Board, cur_player)
    action = SELECTION(board, cur_player)
    board_new, cur_player = GetNextState(board, cur_player, action)
    SIMULATE(board_new, cur_player)                          ▷ Recursive call
end procedure

procedure SELECTION(Board, cur_player)
    Input: Board = s ∈ S, Current Player
    Output: Action
    Initialize: UCB(s), s ∈ S with zeros
    valid_actions = GetValidActions(init_board)
    for action in valid_actions do
        board_new = GetNextState(board, cur_player, action)
        Calculate UCB(board_new)
    end for
    Action = argmax_a UCB(·)
end procedure

procedure EXPAND(Board, cur_player)
    Input: Board = s ∈ S, Current Player
    Output: Initialize count(s), V(s) & append in state_memory
    valid_actions = GetValidActions(init_board)
    for action in valid_actions do
        board_new = GetNextState(board, cur_player, action)
        if board_new is not in state_memory then
            Append in state_memory
            count(s), V(s) = 0, 0
        end if
    end for
end procedure
```

Algorithm 2 MCTS Part 2

```
procedure UPDATE(Board)
    Input: Board = s ∈ S
    Output: Updated V(s)
    for state s in trajectory do
        count(s) ← count(s) + 1
        R = GetReward(s)
        V(s) ← V(s) + (R − V(s))/count(s)
        if board_new not in state_memory then
            Append in state_memory
            count(s), V(s) = 0, 0
        end if
    end for
end procedure

procedure ROLLOUT(Board, cur_player)
    Input: Board = s ∈ S, Current Player
    Output: V(s)
    init_board = Board
    s_init = init_board
    action = RandomAction(Board)
    board, cur_player = GetNextState(board, cur_player, action)
    while game is not ended do
        action = RandomAction(board)
        board, cur_player = GetNextState(board, cur_player, action)
    end while
    R = GetReward(board)
    V(s_init) = R
end procedure

procedure PLAY(Board)
    Input: Board = s ∈ S (from Arena)
    Output: Action (for Arena)
    Initialize: state_memory as empty list
    cur_player=1
    for iteration = 1, 2, ..., N do
        Initialize: trajectory as empty list
        SIMULATE(board, cur_player)      ▷ Grow the tree using SIMULATE() function
    end for
    Action = argmax_a V(s)
end procedure
```

Algorithm 3 Q-Learning Part 1

procedure TRAIN()
Input: Learning rate, α; $\epsilon > 0$; discount factor, $\gamma \in [0, 1]$; exploration discount factor, dc; min. value of ϵ, $e\_min > 0$
Output: Q-Table
init_board = InitializeBoard() ($=s_0$)
for episode $= 1, 2, \ldots, N$ **do**
Board = init_board
if episode $= \frac{2}{3}\cdot$ N **and** $dc = 1$ **then**
$e = 0$
end if
if episode mod 2 $= 0$ **then**
cur_player = 1
else
cur_player = -1
OPPONENTPLAY(Board, cur_player)
end if
INITQTABLE(Board)
while game is not ended **do**
action = ϵ-GREEDY(Board)
Board, cur_player = GetNextState(Board, cur_player, action)
CHECKTERMINAL(Board)
Board, cur_player = OPPONENTPLAY(Board, cur_player)
CHECKTERMINAL(Board)
INITQTABLE(Board) ▷ Board $= s \in \mathcal{S}$
max_q_prime = $\max_a Q(s', a)$
UPDATE(R=0, max_q_prime)
end while
end for
end procedure

procedure OPPONENTPLAY(Board, cur_player)
Input: Board, cur_player
Output: next_board, next_player
action = RandomAction(Board)
next_board, next_player = GetNextState(Board, cur_player, action)
end procedure

procedure INITQTABLE(Board)
Input: Board $= s \in \mathcal{S}$
Output: $Q(s, a)$ initialized with zeros (if not already)
valid_actions = GetValidActions(Board)
for action in valid_actions **do**
a = action
if $Q(s, a)$ not in Q-Table **then**
$Q(s, a) = 0$
end if
end for
end procedure

Algorithm 3 Q-Learning Part 2

```
procedure ϵ-Greedy(Board)
    Input: Board = s ∈ S
    Output: Action
    Generate random number m ∈ [0, 1]
    if m < e then
        action = RandomAction(Board)
        if e > e_min then
            e = e · dc
        end if
    else
        action = argmax_a Q(s, a)
    end if
end procedure

procedure CheckTerminal(Board)
    Input: Board = s ∈ S
    Output: Call Update() if the game is ended
    if game is ended then
        R = GetReward(Board)
        max_q_prime = 0
        Update(R, max_q_prime)
    end if
end procedure

procedure Update(R, max_q_prime)
    Input: Immediate reward R; max_q_prime (= max_a Q(s', a))
    Output: Updated Q(s, a) value
    Q(s, a) ← Q(s, a) + α(R + γ · max_q_prime − Q(s, a))
end procedure

procedure Play(Board)
    Input: Board = s ∈ S (from Arena)
    Output: Action (for Arena)
    Action = argmax_a Q(s, a)
end procedure
```

B Plots for TicTacToe 5×5, Connect4 (6×7, 4)

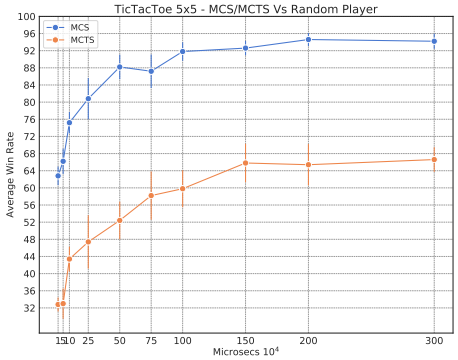

Fig. 11: MCS and MCTS average win–rates for 5 different seeds, at TicTacToe 5×5

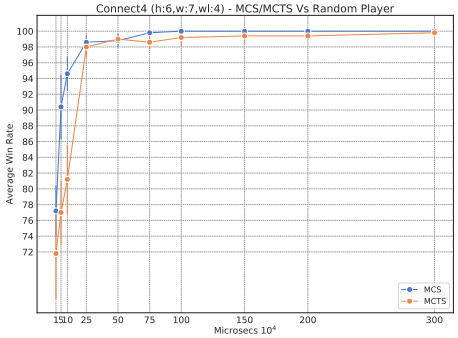

Fig. 12: MCS and MCTS average win–rates for 5 different seeds, at Connect4 (6×7, 4)

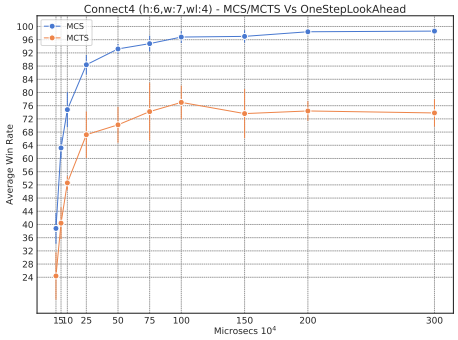

Fig. 13: MCS and MCTS average win–rates against OneStepLookAhead for 5 different seeds, at Connect4 (6×7, 4)

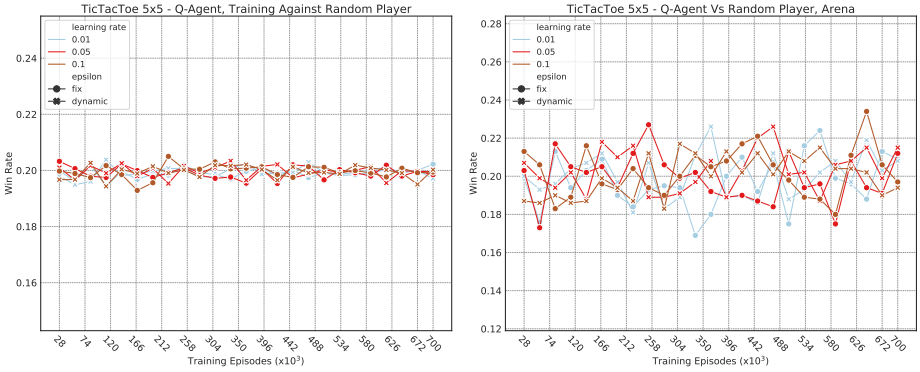

Fig. 14: TicTacToe 5×5. Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

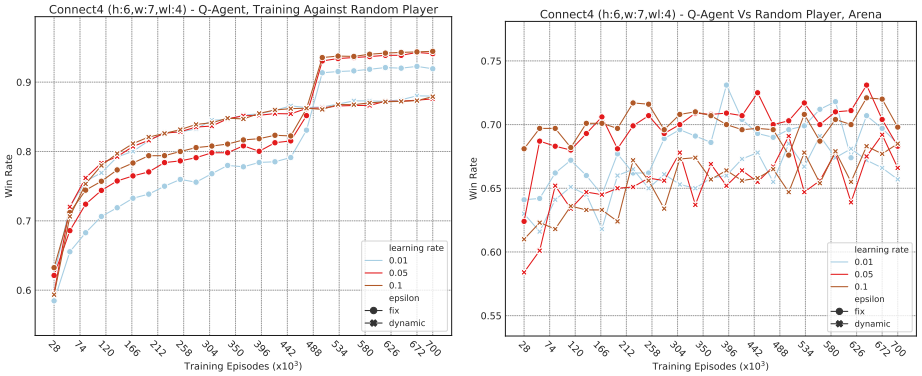

Fig. 15: Connect4 (6×7, 4). Left: Win–rate of Q–Agent against the random player in the training procedure. Right: Average win–rate of Q–Agent against the random player in Arena.

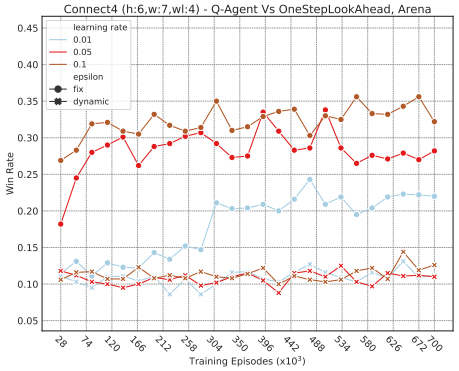

Fig. 16: Win–rate of Q–Agent against OneStepLookAhead in Arena at Connect4 (6×7, 4).