ULTIMATE TIC-TAC-TOE: A REINFORCEMENT LEARNING APPROACH

A REPORT

Cesare Spinoso-Di Piano
Department of Arts and Sciences
Concordia University
1455 Boulevard de Maisonneuve O, Montreal
Quebec, Canada
cesare.spinoso@gmail.com

Mathieu Dugré
Gina Cody School of Engineering and Computer Science
Concordia University
1455 Boulevard de Maisonneuve O, Montreal
Quebec, Canada
mathieu.dugre@mail.concordia.ca

August 24, 2020

ABSTRACT

The game of tic-tac-toe is a well-solved problem. However the game has multiple variant some of them offering a much higher level of difficulty. The variant *Ultimate tic-tac-toe (UTTT)* is particularly complex for human player as the combinatoric of game possibility is enormous. Computer have proven their success in different game with similar issues. We propose to train a *reinforcement learning* agent to play the game. In this report, we analyze how a reinforcement learning approach can be used to handle the Ultimate Tic-Tac-Toe game. In particular, we discuss our results of employing Q-Learning as well as constructing features and using One-Step Sarsa (Semi-Gradient).

***K*eywords** Ultimate Tic-Tac-Toe · Q-Learning · Double Q-Learning · One-Step Sarsa (Semi-Gradient)

1 Introduction

The Markov decision process (MDP) we consider in this report is the game Ultimate Tic-Tac-Toe. In Ultimate Tic-Tac-Toe, two players play on a 9×9 board consisting of 9 regular 3×3 Tic-Tac-Toe boards. Much like in regular Tic-Tac-Toe, a player starts by playing "X" in any spot on the board. Then player "O" plays and this continues until a player wins or the game ends in a draw. The main differences between regular and ultimate tic-tac-toe are twofold. Firstly, when a "sub" Tic-Tac-Toe board fills up it is either a draw, goes to "X" or goes to "O". In the case that either "X" or "O" wins this sub-board, a bigger "X" or "O" is drawn to illustrate that this sub-board belongs to its winner (Figure 1). The game is ultimately won or lost by having three of the same symbol in the larger Tic-Tac-Toe boards (horizontally, vertically or diagonally). The second difference is that the sub-board on which for instance "O" plays is determined by where "X" plays on its previous turn. For example, if player "X" chooses to play in the *top-right* tile of the middle sub-board, then "O" will need to play its next move in the *top-right* sub-board (Figure 2). Moreover, in the case that this top-right sub-board was already won by either "X" or "O", the player that plays next is allowed to place his symbol *anywhere* on the whole board. This is what gives Ultimate Tic-Tac-Toe an additional layer of complexity over the original game.

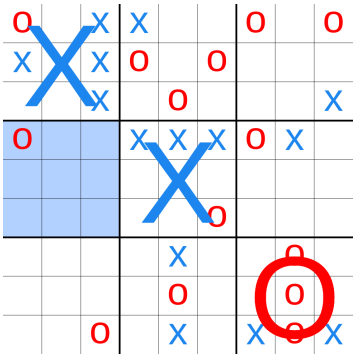

Figure 1: When a sub-board is won, it is marked by the player's symbol ("X" or "O").

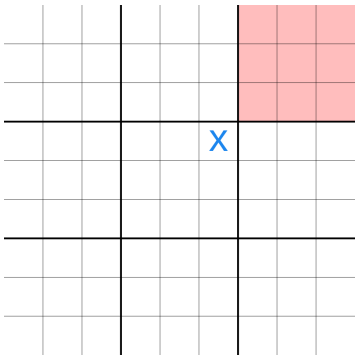

Figure 2: First player plays "X" in the top-right tile of the middle board. This forces the player who plays "O" to play in the top-right board.

The methods we use to tackle this Markov decision process are the Q-Learning [1] method, the double Q-Learning method as well as the use of features and the One-Step Semi-Gradient Sarsa algorithm. In the Q-Learning and double Q-Leaning methods we find an approximation to the optimal action-value q_* of a reduced state space that tries to replicate the full state space of the game. In the case of features, we maps state-action pairs to a set of features and use this construction as a non-tabular approximation for the state-action value function. We then improve this approximation using One-Step Semi-Gradient Sarsa with an ε-greedy heuristic.

There haven't been many accomplished research articles on this topic in particular. The few articles that have researched this topic explore methods such as the minimax algorithm, Deep Q-Learning and Monte Carlo Tree Search (MCTS). For instance, a team of students at Standford University [2] used an adversarial zero-sum game to represent the Ultimate Tic-Tac-Toe game dynamic and used the minimax algorithm to implement an agent which would play effectively against an opponent that plays randomly. They also used Deep Q-Learning and MCTS implementations for their agent. For an agent always playing first against a random opponent [1], the minimax algorithm worked best against a random player (0.97 win rate) although the MCTS was only slightly worse (0.96 win rate). Surprisingly, the Deep Q-Leaning approach was terrible compared to the other two methods (0.58 win ratio).

2 Problem Description

In this MDP, the objective is to get the maximum possible reward by transitioning from one state to another through actions. If we consider the full state space of this problem, the set of states S is the set of all possible combinations of "X", "O" and blank spots on the playing board. Since there are 81 tiles in the board (9×9), the total number of states is $|S| = 3^{81}$. The set of actions $A(s)$ (i.e. where a player may place their symbol "X" or "O") which depends on the current state s is the set of all possible moves that a player can take given a particular board configuration. Usually the size of the set of actions will be at most 9 since a Tic-Tac-Toe sub-board is a 3×3 board. However, there can be state-action transitions that lead to the set of actions being the placement of "X"/"O" on any blank space in the full board. Finally, for the set of rewards R, a reward of -100 is given for a loss, 0 for a draw and 100 for a win. Given the nature of our game, a reward is only received at the end of the episode. This is because an agent is only rewarded for the final outcome; winning, loosing or drawing. We chose not to reward the agent for intermediate goals as this may drive the agent to a sub-optimal objective or policy. Thus the sequence of state, actions and rewards in this MDP is as follows:

$$S_0 \to A_1 \to S_1 \to S_2 \to ... \to S_{T-1} \to A_T \to S_T \to R \tag{1}$$

Where S, A, R, T denote the observed state, action and reward respectively and where T is the terminal time step i.e. the time at which the episode ends. The initial state S_0 is the initial empty Tic-Tac-Toe board. From this state, the agent takes an action A_1. Then the agent's opponent (in this case player "O") makes a move in the new board which becomes S_2. Notice that, for simplicity, we do not consider the opponent's actions as part of the sequence of actions in the MDP but rather part of the states. Also, for the purposes of this report, we will be creating an agent that always plays "X", i.e. that always plays first. A similar analysis could be done for an agent that plays "O". Given the size of our state-space, traditional dynamic programming and look-up table approaches cannot be used to solve a game like Ultimate Tic-Tac-Toe. This is why we choose to use state reductions and non-tabular learning approaches.

[1] We denote here and for the rest of the report a random opponent as an opponent that selects actions at random.

3 Analyses

In this section we analyze the different reinforcement learning techniques that we used to implement an agent that should play against a random opponent.

3.1 Random Agent

The first agent we implemented was an agent that randomly samples actions from a list of valid actions at each given state. This is essentially the same policy as the opponent's random sampling strategy. This allows us to determine a baseline to compare with the methods that we will analyze next. To evaluate the performance of an agent's random policy, we created the function `SimulUTT` which simulates a single episode of Ultimate Tic-Tac-Toe and returns the winner (0 for a draw, 1 for player "X" and 2 for player "O"). By simulating 10000 episodes of this game, the win, draw and loss ratios converged to 0.4, 0.25 and 0.35 (Figure 3).

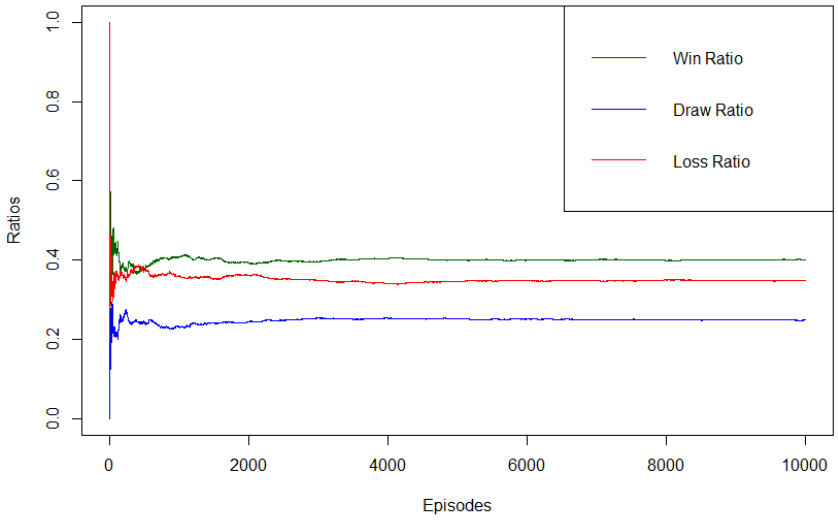

Figure 3: The win, draw and loss ratios when the agent uses a random sampling of valid moves as the policy.

3.2 Q-Learning with State Reduction (1)

As the title suggests, in this method we reduce our full state space to a state approximation which should (in theory) model the full state space but with a smaller space complexity. To reduce the state space, we choose to represent the space set $S' \subseteq S$ as the set of possible combinations of "X","O" and blank spaces (represented here as 1, 2 and 0 respectively) of a single Ultimate Tic-Tac-Toe sub-board. This sub-board is the current sub-board the agent is in, right before it plays "X" in one of the tiles. This state approximation clearly has several limitations since it does not take into consideration whether other sub-boards have been won and what the state of other sub-boards are. Moreover, the state approximation cannot handle cases where the agent has the possibility to play anywhere on the board. In fact, it is essentially the state space of a normal Tic-Tac-Toe MDP. Nevertheless, since the state space is not big ($|S'| = 3^9$) it is possible to use a look-up table to store all the approximations of the state-action values $Q(s, a)$ that we will use to train the agent. The way we map the state S' to the real numbers $\mathbb{R}$ is by using the ternary numerical system as a representation of the sub-board and converting from ternary to decimal. For example:

$$\begin{array}{|c|c|c|}\hline 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = (210\ 010\ 201)_3 \qquad \text{Ternary representation of the board}$$

$$= 2 \times 3^8 + 1 \times 3^7 + 0 \times 3^6 + ... + 0 \times 3^1 + 1 \times 3^0$$

$$= 15409$$

So in general a state vector **x** can be represented as follows in decimal form:

$$\mathbf{x} = \begin{array}{|c|c|c|}\hline x_0 & x_1 & x_2 \\ \hline x_3 & x_4 & x_5 \\ \hline x_6 & x_7 & x_8 \\ \hline \end{array} = \sum_{i=0}^{8} x_i 3^i$$

This allows us to uniquely map every state to a real number which we then use as the index for our look-up table. To map the actions to the real numbers $\mathbb{R}$, we ignore the case where the agent can play in a sub-board other than the one it is currently in. This allows us to map the actions to numbers from 1 to 9 as follows:

1	2	3
4	5	6
7	8	9

Figure 4: The actions that the agent can take in the sub-board are numbered from 1 to 9.

So if the agent "wanted" to play in the middle tile of its sub-board then this would correspond to taking action 5 in the state s (represented previously).

3.2.1 Q-Learning

Now that we have set-up the approximated state space and action space, we are ready to discuss the reinforcement learning method used to train our agent. In order for the agent to play the "best" move a given that it is in a state s we should use the state-action value $Q(s, a)$ which gives the value or expected return of taking action a in state s. The goal is to $Q(s, a)$ approximate the optimal state-action value function q_*. To do so we simulate 10000 runs where the policy $\pi(a|s)$ gives every valid action a in s an equal probability of being selected (i.e. random sampling). We then used the sequence of states, actions and reward (1) from every episode to improve our estimate $Q(s, a)$ of the optimal action-value function $q_*(s, a)$ using the Q-Learning temporal difference approach where:

$$Q_{t+1}(S_t, A_t) := Q_t(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q_t(S_{t+1}, a) - Q_t(S_t, A_t)\big]$$

Where in our case $\gamma = 1$ and we use the step size $\alpha = 0.1$. To evaluate this learning method, we run the same 10000 episode simulation except that in this simulation the agent uses the values in the state-action value table $Q(s, a)$ to choose the action to take. From a list of valid moves, the agent chooses the one that maximises $Q(s, a)$ i.e.

$$A_{t+1} := \operatorname*{argmax}_a Q(S_t, a)$$

To handle the special case where the agent has the choice to play on more than one sub-board, we choose to randomly sample what sub-board the agent should pick (i.e. we randomly choose the state s the agent has transitioned to) and then proceed to pick the action the same way.

As a result of the simulation, the new win, draw and loss ratios are 0.44, 0.02 and 0.55 respectively (Figure 5). Thus, although we have improved the win rate by 4%, the draw rate has decreased to almost 0% and the loss rate has actually surpassed the win rate by 11%. This may be explained by the fact that, despite our use of the estimate state-action value $Q(s, a)$, the agent picks most of its actions at random. This hybrid between a random policy and a sub-optimally reduced state space explains in part why the loss rate has surpassed the win rate.

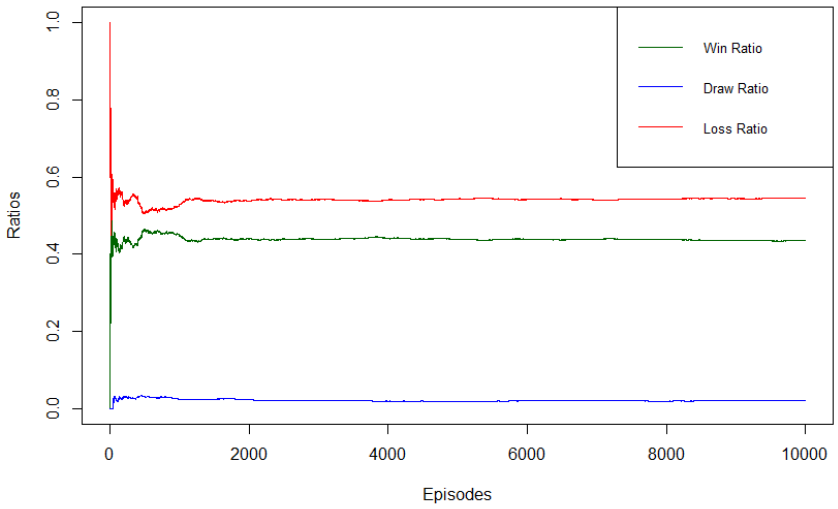

Figure 5: Win, loss and draw ratios when using $Q(s, a)$ from Q-Learning and the reduced state-space.

Clearly, this was not the outcome we hoped for, but it nevertheless yields interesting results. As mentioned previously, one would expect this method to be very similar to the random policy since the state-action value $Q(s, a)$ is not a very good state approximation as it only focuses on its sub-board (or state subset) rather than the entire board (state space).

3.3 Q-Learning with State Reduction (2)

The state approximation we chose in the first part was clearly not sufficiently adequate to allow the agent to learn an efficient deterministic policy, the loss ratio surpassing the win ratio. Since the win ratio did increase with respect to the baseline win ratio, this suggests that a better state approximation could lead to a better win ratio. To improve our state reduction, we increase its size by a factor of 9 by adding a component that gives the location of the current sub-board. That is, in the previous state approximation we only had the current sub-board as a state approximation. Now, we are using the current sub-board plus its "location" which we describe as follows:

1	2	3
4	5	6
7	8	9

Figure 6: The "locations" of the sub-boards numbered from 1 to 9.

Thus, the state set S looks like a pair of elements $(\mathbf{x}, y)$ where $\mathbf{x} = [x_0, x_1, ..., x_8]$ is the ternary representation of the board described previously and y is the location of that sub-board (Figure 6). Thus to uniquely map the reduced state set $S' \subseteq S$ to the real numbers $\mathbb{R}$ we combine the previous conversion from ternary to decimal with a shift using y and a non-overlapping constant (3^9 for simplicity). A conversion in the general case is described as follows:

$$\Big(\mathbf{x}, y\Big) = \Big(\begin{array}{c|c|c} x_0 & x_1 & x_2 \\ \hline x_3 & x_4 & x_5 \\ \hline x_6 & x_7 & x_8 \end{array}, y\Big)$$

$$= \sum_{i=0}^{8} x_i 3^i + (y-1)3^9$$

This state representation increases the state size from 3^9 to 3^{11}, increasing the space complexity by a factor of 9. However, as our results will soon show, this is an appropriate trade-off to get a more representative state-action value approximation. The actions still remain the possible spots on the sub-board where the agent may place its symbol. The mapping to decimal is thus identical to the one in the previous reduced space state (Figure 4).

3.3.1 Q-Learning

The first reinforcement learning technique was the same Q-Learning approach described in the first state reduction with only the current sub-board. Using this new state approximation, the win, draw and loss ratios converged to 0.45, 0.23 and 0.32 respectively (Figure 7). This is definitely an improvement since the percentage of games lost is no longer greater than the percentage of games won. Moreover, the win ratio has improved 5% with respect to the baseline win ratio.

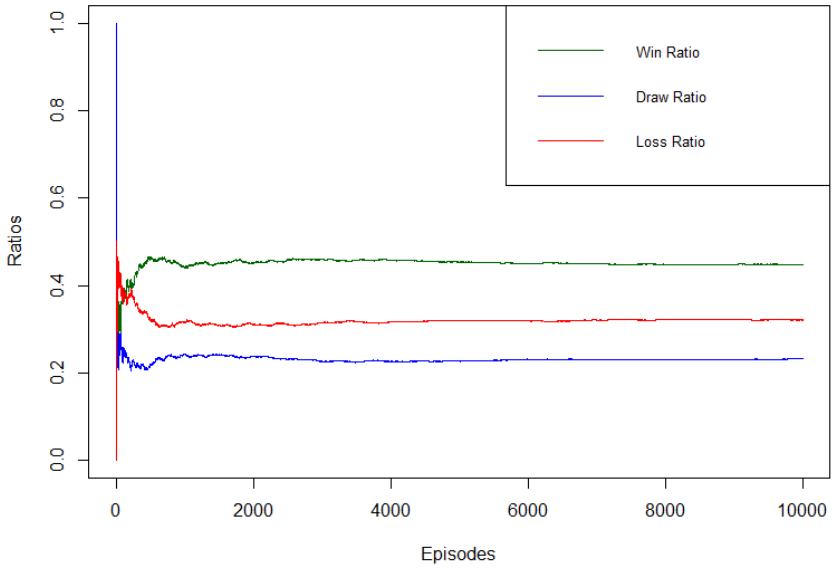

Figure 7: Win, draw and loss ratio when using $Q(s, a)$ from Q-Learning and the improved state approximation.

3.3.2 Double Q-Learning

Since this state approximation yielded better results than the first, we decided to further investigate using another reinforcement learning technique. Similar to Q-Learning, double Q-Learning is a temporal difference method that used two action-state function $Q_t^1(s, a), Q_t^2(s, a)$ instead of one. Only one (by convention $Q_t^1(s, a)$) is used to select the optimal action given a state s, but both are improved during the learning phase. The goal of double Q-Learning is to reduce the bias that is introduced from using the same data to determine the optimal action and its value [3]. In double Q-Leaning, the update of the state-action function requires a coin toss. If the coin falls on "heads" we follow the heuristic that we update $Q_{t+1}^1(s, a)$ and keep $Q_{t+1}^2(s, a)$ unchanged from its previous iteration. This would look like the following:

$$Q_{t+1}^1(S_t, A_t) := Q_t^1(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \max_a Q^2\big(S_{t+1}, \operatorname*{argmax}_a Q_t^1(S_{t+1}, a)\big) - Q_t^1(S_t, A_t)\Big)$$
$$Q_{t+1}^2(S_t, A_t) := Q_t^2(S_t, A_t)$$

Where $\gamma = 1$ and the step size $\alpha = 0.1$. If the coin had landed on "tails", then the roles of $Q_t^1(s, a), Q_t^2(s, a)$ would be swapped in the above assignment operations.

As a result of this method, the win, draw and loss ratios converged to 0.55, 0.08 and 0.37 respectively (Figure 8). This is a significant improvement over the baseline win ratio (approximately 15%). Thus, the reduction of the state-action value bias significantly improved the agent's performance.

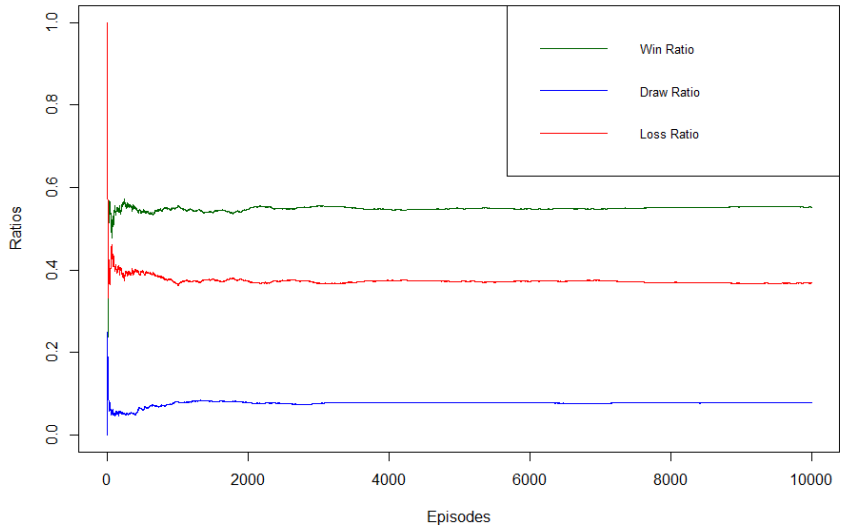

Figure 8: Win, draw and loss ratio when using $Q(s, a)$ from double Q-Learning and the improved state approximation.

3.4 Q-Learning with State Reduction (3)

For the third state approximation, we use the previous state approximation but chose to also include in $S' \subseteq S$ a third element $\mathbf{z} = [z_0, z_1, ..., z_8]$ which would track the sub-boards that have been won by "X". That is, if, for instance, the agent was in the top-left sub-board ($y = 1$) with a certain configuration $\mathbf{x}$ of that sub-board, then $\mathbf{z}$ would track which sub-boards it won so far. For instance, if the top-right board had been previously won by "X" then it (z_2) would be marked as 1.

$$\mathbf{z} = \begin{array}{|c|c|c|}\hline z_0 & z_1 & z_2 \\ \hline z_3 & z_4 & z_5 \\ \hline z_6 & z_7 & z_8 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ \hline \end{array}$$

Figure 9: The $\mathbf{Z}$ component indicates which of the sub-boards "X" has won. In this instance "X" has won the left-middle sub-board and the two bottom-right sub-boards.

Because of memory limitations, the third variable $\mathbf{z}$ does not distinguish between a "blank" sub-board (i.e. a sub-board that hasn't been won or a sub-board that has ended in a draw) and a sub-board won by "O". As a result, the size of the state subset S' is now $|S'| = 3^{11} * 2^9$. If we include the 9 spaces required for every action in $Q(s, a)$, this is by far the largest look-up table used so far taking about 6.1 GB of memory. As we can observe, the better we try to make our state approximation, the more information we need, the larger the look-up table becomes. For the mapping from S' to the set of real number $\mathbb{R}$ we use an equation similar to the previous two instances. The mapping in general is as follows:

$$\begin{aligned}(\mathbf{x}, y, \mathbf{z}) &= \left(\begin{array}{|c|c|c|}\hline x_0 & x_1 & x_2 \\ \hline x_3 & x_4 & x_5 \\ \hline x_6 & x_7 & x_8 \\ \hline \end{array}, y, \begin{array}{|c|c|c|}\hline z_0 & z_1 & z_2 \\ \hline z_3 & z_4 & z_5 \\ \hline z_6 & z_7 & z_8 \\ \hline \end{array} \right) \\ &= \sum_{i=0}^{8} x_i 3^i + (y-1)3^9 + (\sum_{i=0}^{8} z_i 2^i)3^{11}\end{aligned}$$

Given that this state approximation provides more information, one would expect that it should produce a better agent. However, as we will explain in the next section, the impracticality of such a big look-up table (over 6 GB of memory) causes significant drops in the learning rate. Moreover, because of the size of the look-up table, we deemed it infeasible to try to use double Q-Learning as before since this would require twice as much memory (using two state-action value functions).

3.4.1 Q-Learning

The only technique we used given the size of our look-up table was the Q-Learning similar to the two previous state approximations. The win, draw and loss ratios of our agent using the approximated $Q(s, a)$ converged to 0.43, 0.25 and 0.32 respectively (Figure 10).

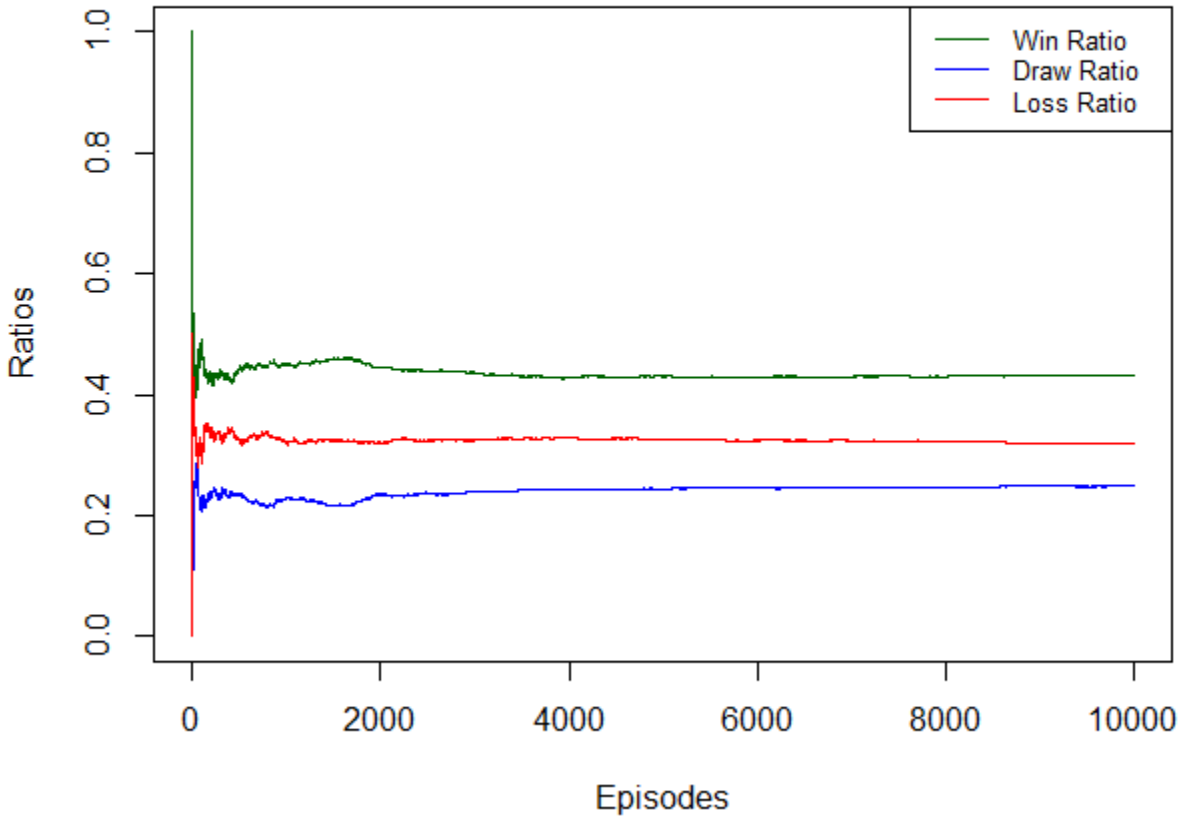

Figure 10: Win, draw and loss ratio when using $Q(s, a)$ from Q-Learning and the third state approximation.

This is surprising. Even though our state approximation used more information than the previous one, the win ratio was actually lower than the second state approximation that only used the current board and the location of that current board. The win ratio was higher than the random baseline however it was worse than previous Q-Learning methods. This suggests that a state approximation construction is not the best way of tackling this problem. We will explore a non-tabular approach next.

3.5 Features and Semi-Gradient Sarsa

In this non-tabular approach, we use features to map state action pairs (s, a) to a smaller state-space. For our particular application, the feature $\mathbf{x}(s)$ belongs to $\mathbb{R}^{20}$ (i.e. 20 dimensions). We decided to construct the feature intuitively by setting the first nine components $(x_1...x_9)$ to the number of "X"s in each sub-board (where the location follows the indexing convention used previously - Figure 6). The next nine components $(x_{10}...x_{18})$ correspond to the number of "O"s in each sub-board. Finally, the last two components, x_{19}, x_{20} correspond to the number of sub-boards won by "X" and "O" respectively. Since we need to take into account actions (as we are trying to improve our agent), $\mathbf{x}(s, a)$ is constructed the same way as $\mathbf{x}(s)$ except that everything is counted *after* action a has been taken.

To estimate and improve $q_\pi(s,a)$, we use a parametric approximation $\hat{q}(s,a,\mathbf{w})$ where the parameter $\mathbf{w}$ is a weight vector and has the same dimension as $\mathbf{x}$. For simplicity, we construct $\hat{q}(s,a,\mathbf{w})$ as the linear combinations of the weight and feature vector. That is:

$$\hat{q}(s,a,\mathbf{w}) = \mathbf{w}^T\mathbf{x}(s,a) = \sum_{i=1}^{20} w_i x_i(s,a)$$

Then, the gradient with respect to the weights easily simplifies as follows:

$$\begin{aligned}
\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w}) &= \big[\frac{\partial\hat{q}(s,a,\mathbf{w})}{\partial w_1},...,\frac{\partial\hat{q}(s,a,\mathbf{w})}{\partial w_{20}}\big]^T \\
&= \big[\frac{\partial\sum_{i=1}^{20} w_i x_i(s,a)}{\partial w_1},...,\frac{\partial\sum_{i=1}^{20} w_i x_i(s,a)}{\partial w_{20}}\big]^T \\
&= [x_1,...,x_{20}]^T \\
&= \mathbf{x}(s,a)
\end{aligned}$$

3.5.1 Semi-Gradient One-Step Sarsa

We use the One-step Sarsa semi-gradient method to improve the weights $\mathbf{w}$ of our approximation $\hat{q}(s,a,\mathbf{w})$. The goal of this method is to for our estimate to be very close to the optimal state-action value function ($\hat{q} \approx q_*$). The update step for the weights is as follows:

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\big[R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t) - \hat{q}(S_t,A_t,\mathbf{w}_t)\big]\nabla\hat{q}(S_t,A_t,\mathbf{w}_t)$$

Where S_t, A_t are the states and actions at time step t and S_{t+1}, A_{t+1} are the states and actions at time step $t+1$. As previously, the discount factor $\gamma = 1$ since this is an undiscounted MDP. Also, we observed that without a small enough step size factor α the weights $\mathbf{w}$ would increase too quickly. As a result, we use $\alpha = 2^{-20}$.

To select the action at time step t, we choose to use an ε-greedy approach. That is, at time step t the selected action A_t follows the following probability mass function:

$$A_t := \begin{cases} \text{argmax}_{a\epsilon A(S_t)}\,\hat{q}(S_t,a,\mathbf{w}_{t-1}) & \text{probability } = 1-\varepsilon \\ \text{Random Sample from } A(S_t) & \text{probability } = \varepsilon \end{cases}$$

To guarantee convergence, ε should slowly decrease. To do so, we chose the heuristic that ε should be halved at every 1000 time steps[2]. To summarize, this method essentially involves selecting an action a by following an ε-greedy method based on $\hat{q}(.,.,\mathbf{w}_t)$, observing the resulting states s and rewards r and using those values to update the weight $\mathbf{w}_{t+1}$ thus improving our estimate of q_*.

Using $\varepsilon = 0.4$ and simulating 10000 episodes, the win, draw and loss ratios converged to 0.96, 0.00 and 0.04 respectively (Figure 11).

We can observe that for $t < 1000$, the ratios seem to be converging to a a different value than for $t > 1000$. This is due to the heuristic of reducing ε by half at every 1000 time steps. Initially, the agent explores frequently (40% of the time). However, as the weights better approximate q_*, there is less need for exploration and the agent exploits more frequently its estimate of q_*. That is to say that the probability of taking the greedy action converges to 1, which is why we should also expect the win ratio to converge to approximately 1. This is clearly a huge improvement over the look-up table approaches that we previously used.

To further test this method, we used the same set-up except that we changed ε to be slightly smaller. This was in order to test whether or not $\hat{q}$ would converge to q_* given that it would explore less that in the first test where $\varepsilon = 0.4$. For $\varepsilon = 0.2$, the win, draw and loss ratios converged to 0.99, 0.00 and 0.01 respectively. For $\varepsilon = 0.1$, they converged to 0.85, 0.00 and 0.15 respectively. To demonstrate the difference in rate of convergence, we produce a plot of the win ratios for $\varepsilon = 0.4, 0.2, 0.1$ (Figure 12).

[2]This is not a well known or documented heuristic. However, after testing different methods (e.g. decrementing ε at every time step), this one performed the best.

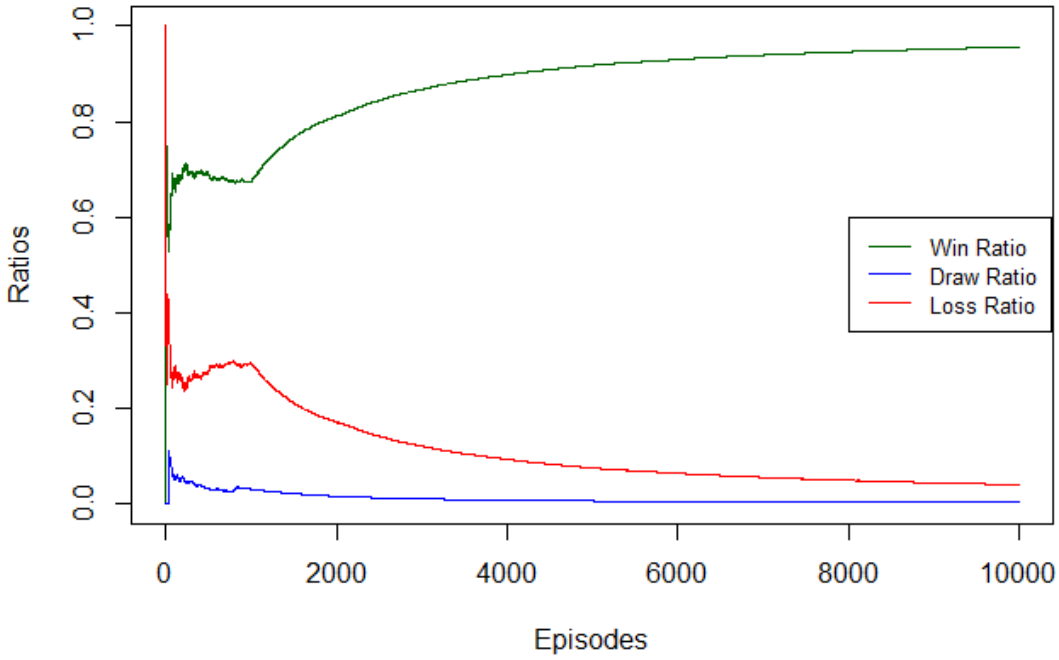

Figure 11: Win, draw and loss ratios using features and One-Step Sarsa (semi-gradient).

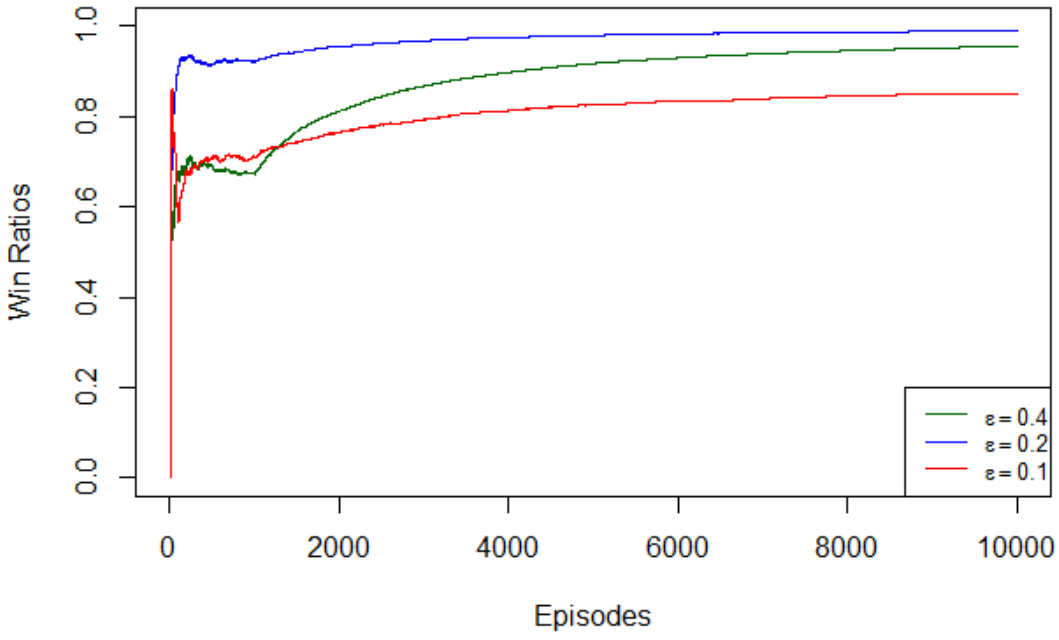

Figure 12: Win ratio for $\varepsilon = 0.4, 0.2, 0.1$ using the feature approximation and One-Step Sarsa (semi-gradient).

As we can observe from the graph, using too small of an epsilon ($\varepsilon = 0.1$) causes the agent to be too greedy and converge to a sub-optimal estimate for q_* (and thus a lower win ratio). On the other hand, using too big of an epsilon ($\varepsilon = 0.4$) causes the agent to explore too much even after its estimate for q_* is quite good which results in a slower convergence rate. Using $\varepsilon = 0.2$ seems to yield the best results as it appropriately trades off exploration and exploitation.

To summarize, using intuitive features and One-Step Sarsa yields much better results than any of the other state approximations with Q-Learning methods. This isn't surprising since non-tabular methods such as this one (i.e. using a parametric approximation of the state-action value function) are known to perform much better than tabular methods for very large state spaces.

4 Conclusion

In conclusion, through various reinforcement learning techniques we tackled the problem of solving the Ultimate Tic-Tac-Toe game as a Markov decision process. Both tabular and non-tabular approaches were used. The non-tabular approaches performed much better given that they did not rely on incomplete or noisy state approximations. The non-tabular methods which included the One-Step Sarsa (semi-gradient) method using a set of intuitively created features worked very well. We summarize our results in the following tables by tabulating the converged win, draw and loss ratio for the tabular and non-tabular methods (Figure 13 and 14). Recall that playing randomly against a random opponent yielded win, draw and loss ratios of 0.4, 0.25 and 0.35 respectively.

	Ratios		
Method	Win	Draw	Loss
State Approx. 1 + Q-Learning	0.44	0.02	0.55
State Approx. 2 + Q-Learning	0.45	0.23	0.32
State Approx. 2 + Double Q-Learning	0.55	0.08	0.37
State Approx. 3 + Q-Learning	0.43	0.25	0.32

Figure 13: Win, draw and loss ratios for the tabular reinforcement learning methods.

	Ratios		
Method	Win	Draw	Loss
One-Step Sarsa $\varepsilon = 0.4$	0.96	0.00	0.04
One-Step Sarsa $\varepsilon = 0.2$	0.99	0.00	0.01
One-Step Sarsa $\varepsilon = 0.1$	0.85	0.00	0.15

Figure 14: Win, draw and loss ratios for the non-tabular reinforcement learning methods.

There is definitely more research to be done in creating and improving agents for this game. For instance, one could consider not only training against a random opponent, but also against a "smarter" one. In this situation, one would probably train the agent to play against a random agent initially and then play against itself. Moreover, an analysis of how playing second could also be interesting as we only focused on agent that would always play first. We suspect that the agent would be at a disadvantage, but it is difficult to say given the difficulty and complexity of this problem.

References

[1] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[2] Phil Chen, Jesse Doan, and Edward Xu. Ai agents for ultimate tic-tac-toe. 2014.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.