UNIVERSITY OF COPENHAGEN

Reinforcement learning techniques for learning Tic-Tac-Toe

Bachelor project in Computer Science

Christian Virt
skm861@alumni.ku.dk

June 17, 2019

Department of Computer Science,
University of Copenhagen

Supervisor:
Jonas Peters
jonas.peters@math.ku.dk

Abstract

By combining a Monte Carlo reinforcement learning technique with a non-contextual stochastic gradient ascent decision policy, we are able to construct a Monte Carlo stochastic gradient ascent policy. This Monte Carlo stochastic gradient ascent policy is able to approximate an optimal strategy in Tic-Tac-Toe, that achieves a higher cumulative gain than expected. Other policies such as Monte Carlo ϵ-greedy, contextual ϵ-greedy, contextual UCB policy, and non-contextual ϵ-greedy have also been implemented and compared. The policies are compared in terms of computational complexity, adaptability and ability to approximate an optimal Tic-Tac-Toe strategy. Out of all policies implemented, the Monte Carlo stochastic gradient ascent policy is the only policy that approximated an optimal Tic-Tac-Toe policy. However, other policies might have been able to approximate an optimal strategy with more training.

Contents

1 Introduction

In this project, several different reinforcement learning techniques have been implemented and compared, in order to find a technique that will learn the optimal strategy for playing Tic-Tac-Toe. I will compare the different reinforcement learning techniques in terms of computational complexity and ability to adapt to a changing environment. I will also carry out an experiment where the different techniques will play against each other. Hopefully, this project will further develop the knowledge and understanding of reinforcement learning, by applying it in practice.

1.1 Tic-Tac-Toe

The game Tic-Tac-Toe is a 2-player game that uses a square 3×3 game board with a total of 9 playing fields. The first player to get three pieces in a horizontal, vertical or diagonal row wins. See figure 1 for examples. The player that plays first uses cross pieces, and the other player uses circle pieces. Once a piece has been placed in a field that field is locked, such that the other player cannot place a piece in the same field. If no player manages to place three pieces in a row, the game ends in a draw.

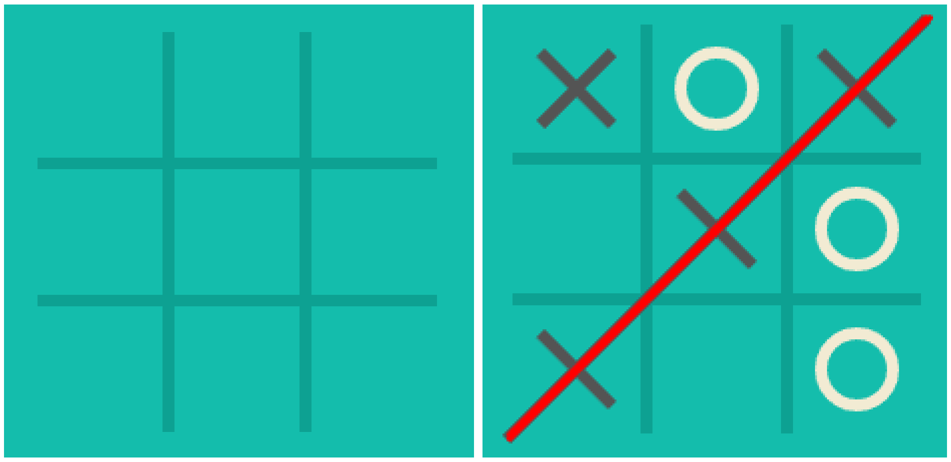


Figure 1: On the left we see an empty Tic-Tac-Toe game board. On the right we see a Tic-Tac-Toe game board where the player that plays first and uses the cross pieces has won.

1.1.1 Implementation of Tic-Tac-Toe

To be able to apply reinforcement learning to Tic-Tac-Toe, I have implemented the Tic-Tac-Toe game in R, as a function called `game`. It uses a helper-function called `checkVictory` to check if a draw, loss or win occurred. The `game` function can take three arguments as input and has the API

```
game = function(player1, player2, switch = FALSE)
```

where the `player1` and `player2` arguments are the concerned players. The `game` function outputs a loss if `player1` loses. If we set the argument `switch = TRUE`, it will instead output a win in the case that `player1` loses. If we call the function `game` with arguments `player1` and `player2`, then `player1` will always play first (cross) and `player2` will play second (circle).

The arguments `player1` and `player2` are essentially data structures that each contain a function for calculation of an action and a data structure used for this calculation.

To make a new player, we simply call the function `newPlayer`, which is defined as:

```
newPlayer=function(inputData,inputAction){
  object=new.env(parent=globalenv())
  object$data=inputData
  object$action=inputAction
  class(object)='pointer'
  return(object)
}
```

As seen in the code above the `newPlayer` function takes two arguments - `inputData` and `inputAction`. The argument `inputData` can be any kind of data used by the argument function `inputAction` to calculate the next action. If we want to make a player that chooses any action uniformly at random, we would implement the following function to give as the inputAction argument:

```
randomPlayerAction = function(gameboard, prev_action, point,
            player, lost = FALSE, draw = FALSE)
{
  actionSet = extractActionSet(gameboard)
  if (length(actionSet) > 1)
  {
    action = sample(actionSet, 1)
  }
  else
  {
    action = actionSet[1]
  }
  action
}
```

The function `randomPlayerAction` is a function that chooses an action uniformly at random from set of possible actions `actionSet`, using the helper-function `sample`. All of the players will use a helper function called `extractActionSet` to know which actions are available at the time. Now that we have defined a function that will choose an action, we can make a new player that plays actions uniformly at random by calling the function `newPlayer`:

```
playerRandom = newPlayer(NULL,randomPlayerAction)
```

Here, we set `inputData = NULL`, since no data is needed to choose an action uniformly at random.

The function `randomPlayerAction` has a lot of superfluous function arguments, since many of the arguments are not needed. We use these arguments for many other kinds of action functions however. All action functions must take the same amount of arguments. The superfluous function arguments could be avoided by utilizing the `inputData` argument better. I decided that keeping the function arguments was the simplest approach however.

1.1.2 Optimal Tic-Tac-Toe strategy

In Figure 2 and 3 on the next page, we see the optimal strategies for a player that plays first and second, respectively. If a player that plays first (using cross pieces) uses the optimal strategy in Figure 2 and if the opponent is playing each action uniformly at random, then the opponent has a 0% chance of winning and a probability of $\frac{1}{8} \cdot \frac{1}{6} \cdot \frac{1}{4} = \frac{1}{192} \approx 0,00521$ for getting a draw. If a player that plays second (using circle pieces) uses the optimal strategy in Figure 3 and if the opponent is playing each action uniformly at random, then the opponent has a 0% chance of winning and a probability of

$$\frac{4}{9} \cdot \frac{1}{5} \cdot \frac{1}{3} + \frac{4}{9} \cdot \frac{6}{7} \cdot \frac{1}{5} \cdot \frac{1}{3} + \frac{1}{9} \cdot \frac{2}{7} \cdot \frac{2}{5} \cdot \frac{1}{3} + \cdot \frac{1}{9} \cdot \frac{2}{7} \cdot \frac{2}{5} + \cdot \frac{1}{9} \cdot \frac{3}{7} \cdot \frac{1}{5} + \frac{1}{9} \cdot \frac{2}{7} \cdot \frac{1}{5} \cdot \frac{1}{3} = \frac{79}{945} \approx 0.084,$$

for getting a draw. I calculated these probabilities using Figure 2 and 3.

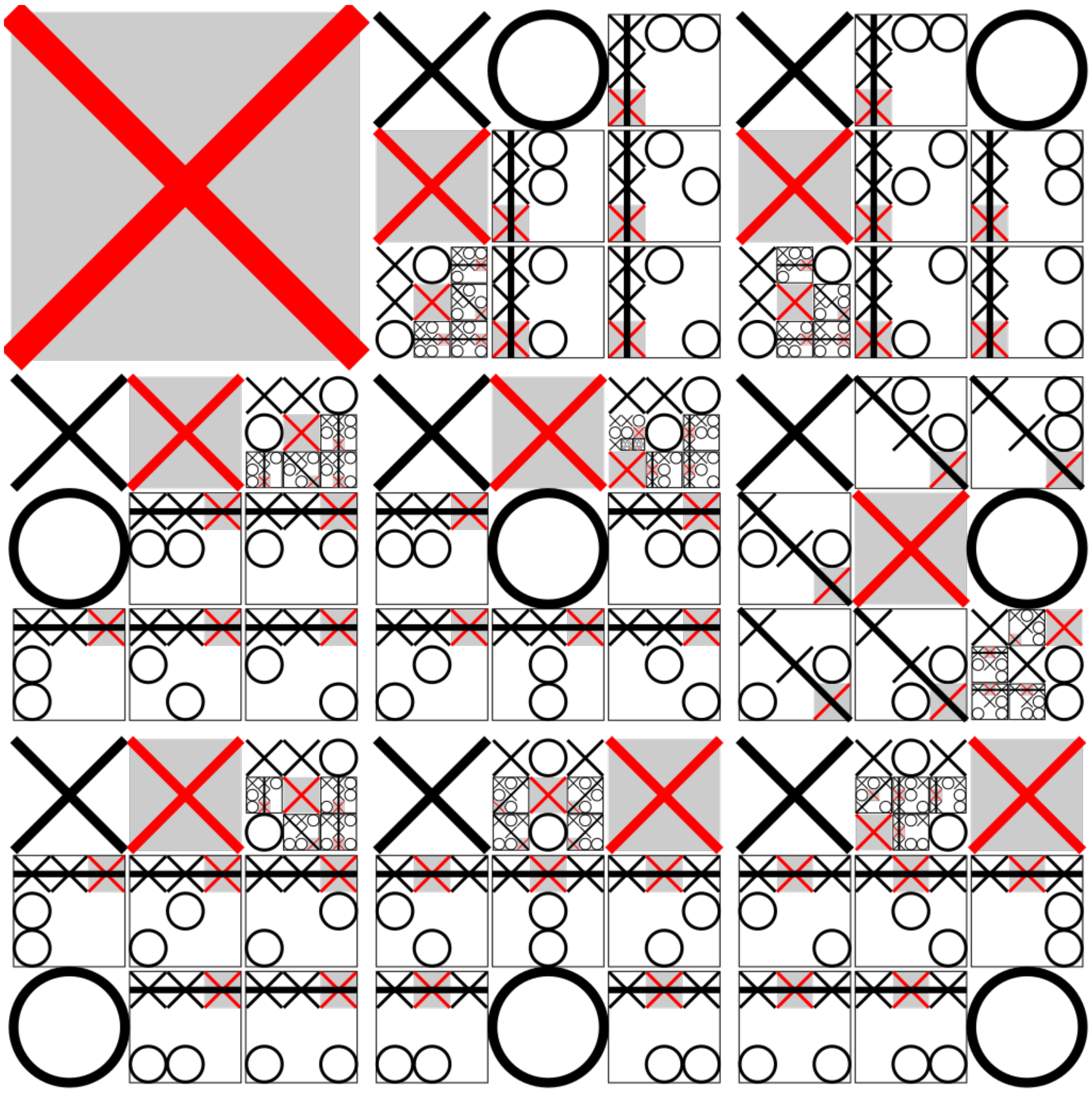


Figure 2: https://commons.wikimedia.org/wiki/File:Tictactoe-X.svg. This is the optimal strategy when playing as cross. Each red cross indicates the action that must be played. For example, our first move should always be in the top-left corner. If the opponent plays in the middle, we will play in the top-middle field as seen according to the strategy. However, if the opponent does not play in the middle, and instead plays in the middle-right field, we will play in the middle. The opponent can play in all fields, where we see a series of smaller game boards. The smaller game boards indicate what actions we are supposed to play in response to the opponents moves. The game boards get smaller for each turn the opponent has played.

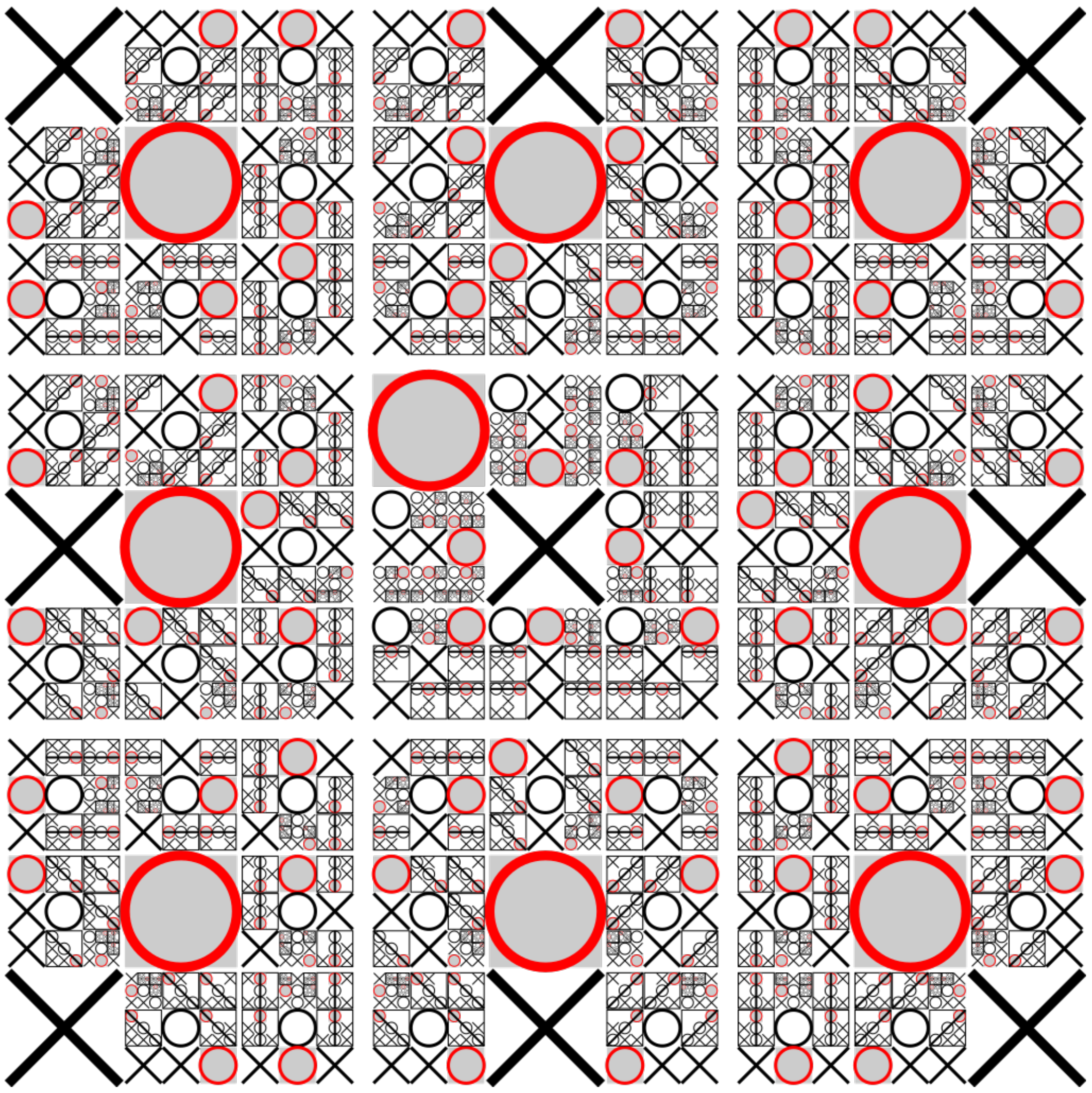


Figure 3: https://commons.wikimedia.org/wiki/File:Tictactoe-O.svg This is the optimal strategy when playing as circle. Each red circle indicates the action that must be played. For example, if the opponent (cross) plays anywhere except for the middle, we play in the middle. Lets assume that the opponent plays cross in the top-left corner. We then play in the middle. Afterwards, the opponent plays in the top-middle. We respond by playing in the top-right. Now the opponent plays in the middle-right, and we win by playing in the bottom-left. For each turn the opponent can play in all fields, where we see a series of smaller game boards. The smaller game boards indicate what actions we are supposed to play in response to the opponents moves. The game boards get smaller for each turn the opponent has played.

1.2 Reinforcement learning

Reinforcement learning is an area in machine learning where a machine has to make decisions and act according to an environment in order to maximize a cumulative gain. In order to make decisions that will maximize a cumulative gain, the machine often has to choose between exploration or exploitation. We might have the knowledge that some action is more rewarding than other actions and therefore more optimal, so we will prefer that action over other suboptimal actions. We want to occasionally explore suboptimal actions however, since suboptimal actions might lead to a higher cumulative gain.

Usually, the more we explore, the closer we get to finding an optimal action that leads to a higher cumulative gain. The environment in a reinforcement learning problem is usually characterised as a Markov decision process (MDP) [5, p. 1-2]. Many modern reinforcement learning techniques are based around the underlying elements of MDPs.

Many of these reinforcement learning techniques are also based around dynamic programming. For many reinforcement learning techniques, dynamic programming is more applicable and efficient than many other problem solving methods [5, p. 11].

1.2.1 The Markov decision process

In a Markov decision process we experience a problem of choosing the action $a \in A$ in an environment (state) $s \in S$ in order to maximize a cumulative gain G. Here, A contains the set of possible actions and S contains the set of possible states. The cumulative gain G is defined later in this section.

We define A_s to be the set of possible actions given a state s. We have a total number of T trials, where s_t is defined as the state at trial t for $0 < t \leq T$. The state s_{t+1} is the state after trial t that results from taking action a_t in state s_t. The conditional probability of transitioning to a state s_{t+1} from state s_t with action $a \in A_s$ at trial t is denoted by

$$\mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a). \tag{1}$$

Transitioning from state s_t to s_{t+1} after performing action a_t at trial t yields the reward

$$r_{a_t}(s_t, s_{t+1}). \tag{2}$$

We generally consider the reward r_{a_t} to be a random variable, since the reward can be random in nature. Tic-Tac-Toe is a deterministic game however and we can therefore consider r_{a_t} as deterministic given a_t, s_t, and s_{t+1}.

The cumulative gain G is defined as a sum of random rewards from each trial `t`:

$$G(T) = \sum_{t=1}^{T} r_{a_t}(s_t, s_{t+1}) \tag{3}$$

The cummulative gain G is a random variable and we want to make the expectation $\mathbb{E}(G)$ as large as possible. We make the expectation of G as large as possible by choosing a policy π that will bring us a maximum possible cumulative gain.

We define s_{t+1} to be a random variable that denotes the state that comes after some action a_t that is performed in the current state s_t [5, p. 37-38]. The decision policy $\pi(s_t)$ is a random variable that for any s_t will generate a realization of some action a_t. A good policy will generate a realization a_t that has the highest possible probability of reaching a state s_{t+1} that will bring us a maximum cumulative gain [5, p. 46].

The name MDP partly stems from the fact that the process is assumed to satisfy the Markov property. If a problem assumes the Markov property in the discrete case, the conditional probability in equation (4) must hold.

Lets assume that we have the random variables $s_1, \ldots, s_t, s_{t+1}$, where each random variable s_i indicates what the state was at trial i for $0 < i < t$. State s_t is the current state, and s_{t+1} is the next state. We also condition on the random variables $a_1, \ldots, a_t$ where each random variable a_k indicates what action was performed at trial k for $0 < k \leq t$.

We also assume that we have a set that contains the states x_j for $0 < j \leq t+1$ and a set that contains the performed actions b_k. The following equation should then hold for all t, for all $x_1, x_2, \ldots, x_{t+1}$, and for all $b_1, b_2, \ldots, b_t$:

$$\begin{aligned} &\mathbb{P}(s_{t+1} = x_{t+1} | s_t = x_t, a_t = b_t, s_{t-1} = x_{t-1}, a_{t-1} = b_{t-1}, \ldots, s_1 = x_1, a_1 = b_1) \\ &\quad = \mathbb{P}(s_{t+1} = x_{t+1} | s_t = x_t, a_t = b_t) \end{aligned} \tag{4}$$

The Markov property in the equation (4) states that we only need to condition on the current state s_t and action a_t to know the conditional probability of the next state s_{t+1} [4, p. 71].

2 Decision policies

I will implement and compare 8 different decision policies. The policies are not my own original ideas, unless specified. Most of the decision policies are taken from Sutton's and Barto's book draft "Reinforcement Learning: An Introduction" [5]. I have however changed some of the notation, since I wanted the notation to be consistent throughout the bachelor thesis. The eight policies are:

1. Left policy
2. Random policy
3. ϵ-greedy policy
4. Contextual ϵ-greedy policy
5. Contextual UCB policy
6. On-policy first-visit Monte Carlo control (using ϵ-greedy)
7. Stochastic gradient ascent policy using inverse probability weighting
8. On-policy first-visit Monte Carlo stochastic gradient ascent

2.1 Left policy and Random policy

For testing purposes, I have implemented a "Left policy". This policy trivially chooses to play from left-to-right and starts with the top row. This policy does therefore not play in the middle row or bottom row, unless the top row is fully occupied. If a field is occupied by another piece, it will skip to the next field. I have also implemented a "Random policy", using the code in section 1.1.1. As mentioned earlier, this policy chooses an action uniformly at random from the set of possible actions.

2.2 ϵ-greedy policy

The ϵ-greedy policy is usually used for context-free K-armed bandit problems. A context-free problem implies that a state s_t, i.e. the environment, does not change depending on the trial t for $0 < t \leq T$, where T is the total number of trials. In other words, all states are equal to each other, meaning that $s_1 = s_2 = \ldots = s_T$.

When all states are equal, it implies that the set of actions A_{s_t} does not depend on s_t and that A_{s_t} therefore remains the same regardless of the state s_t. The Tic-Tac-Toe problem cannot be referred to as a bandit problem, since the set of actions when playing Tic-Tac-Toe depends on the state s_t at trial t [2, p. 662]. In the case of the Tic-Tac-Toe problem we have $A_{s_{t+1}} \neq A_{s_t}$, since there are two less actions in the action set after each trial.

For the ϵ-greedy algorithm we have a fixed value ϵ that determines how often we will choose an action uniformly at random. We will either choose a random action $a \in A$ or we will play the most rewarding action $h \in A$, where A is the set of all actions.

We have to slightly modify the ϵ-greedy algorithm given in Sutton's and Barto's book [5, p. 24], otherwise the policy might play in fields that are already occupied by another piece. We therefore have to check that the actions a and h are in the set of possible actions A_s in state s, if they are not, we have to choose another action. The algorithm assumes that $A_{s_{t+1}} = A_{s_t}$, which is not the case. The decision policy is a stochastic decision policy denoted by

$$\pi(a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A_s|} & , \quad h = a \\ \frac{\epsilon}{|A_s|} & , \quad \text{otherwise} \end{cases},$$

where $|A_s|$ denotes the number of possible actions in state s.

We determine the most rewarding action h by selecting the action $a \in A_s$ that maximizes the action-value $V(a)$. The action-value is defined as the sum of rewards received as a result of performing action a, multiplied by α. The step size α is defined later in this section. $T(a)$ is a random variable that denotes the number of trials where action a was picked. The action-value is denoted by

$$V(a) = \alpha \sum_{i=1}^{T(a)} r_{i,a},$$

where $\alpha = \frac{1}{n}$ or some constant $\alpha \in (0, 1]$ depending on the reinforcement learning problem. Here, $r_{i,a}$ represents the i'th reward received as a result of performing action a.

Since Tic-Tac-Toe is a non-stationary problem, we will choose $\alpha \in (0, 1]$. Non-stationary problems are problems where the reward for performing an action a in a previous state s will impact the return of an action a' in a current state s' and vice versa. I have chosen to set $\alpha = \frac{1}{5}$, since this value of α seems to give the best results based on experimentation and visualization. The constant $\alpha = \frac{1}{5}$, makes the action-value more adaptable, since we give more weight to rewards encountered most recently.

According to Sutton and Barto [5, p. 21], we choose the optimal action h with the formula $h = \text{argmax}_{a \in A} V(a)$. We have to modify this formula for h since the Tic-Tac-Toe game have a varying set of possible actions depending on the state s. Therefore we denote the optimal action h as

$$h = \text{argmax}_{a \in A_s} G(a).$$

2.2.1 Implementation details on the ϵ-greedy policy

The i'th reward for performing action a at trial t where $0 < t, i \leq T(a)$ is denoted by

$$r_{i,a} = \begin{cases} 1 & , \quad \text{action } a \text{ resulted in a win} \\ 0 & , \quad \text{action } a \text{ resulted in an undecided game} \\ -1 & , \quad \text{action } a \text{ resulted in a loss} \end{cases}.$$

The implementation of the algorithm can be seen in the pseudocode below. The pseudocode abstracts away from the implementation details and is heavily inspired by the pseudocode in Sutton's and Barto's book [5, p. 24].

Algorithm 1 ϵ-greedy algorithm

```
Initialize:
for all $a$ from 1 to $|A|$ do
    $V(a) = 0$ //The action-value for some action
    $N(a) = 0$ //The number of times some action was played in some state
end for
Loop:
while true do
    $p =$ uniformly distributed random value between 0 and 1
    $Action =$ action from $A_s$, chosen uniformly at random
    if $p > \epsilon$ then
        $Action = \text{argmax}_{a \in A_s} V(a)$
    end if
    $Reward = Play(Action)$
    $N(a) = N(a) + 1$
    $V(a) = V(a) + \frac{1}{N(a)} \cdot (Reward - V(a))$
end while
```

Computational complexity

When we analyze the computational complexity using asymptotic notation, we see that the initialization process takes $O(|A|)$, where $|A|$ is the total number of actions. In the `while` loop however, the `argmax` will run in $O(|A|)$. The `argmax` function will run for all rounds R that the algorithm plays, resulting in a running time of $O(|A| \cdot R)$.

2.2.2 Performance of the ϵ-greedy policy

We will experiment with a variety of different ϵ values to see how the value of ϵ influence the learning curve and thereby the long-term win rate. We will try with $\epsilon \in \{0.01, 0.05, 0.1, 0.15\}$, to see if we can find a tendency.

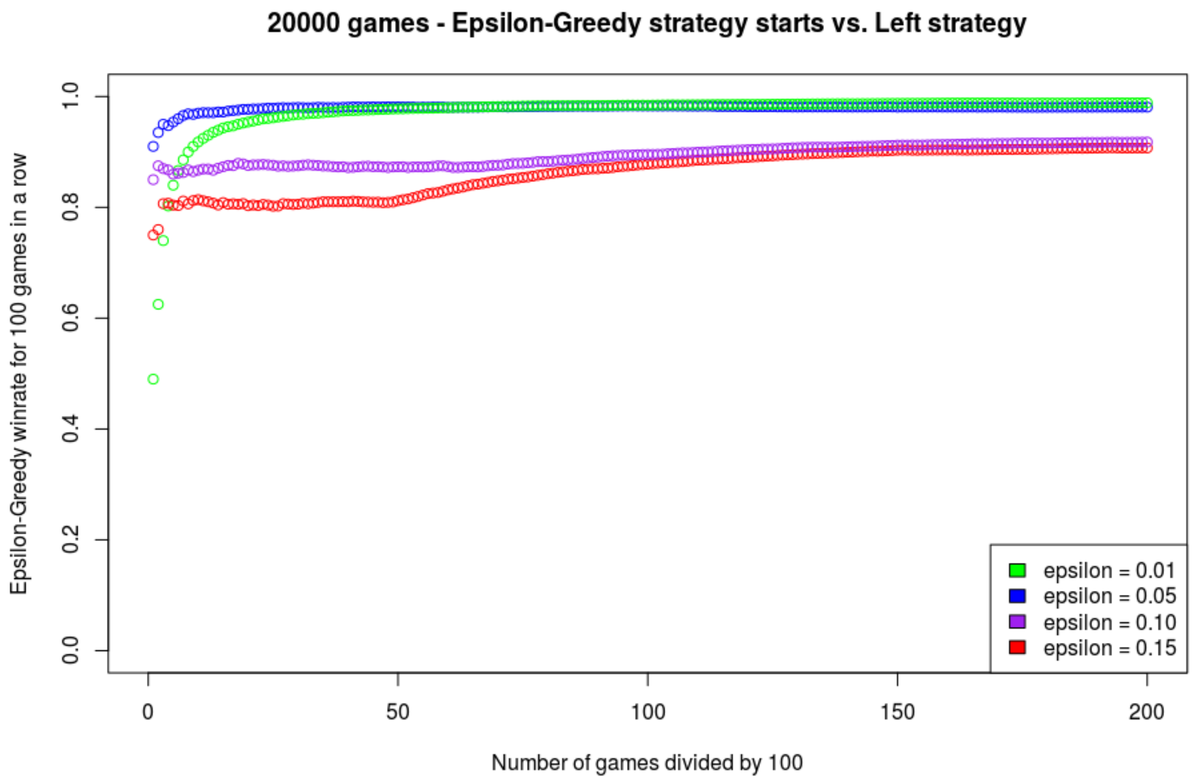


Figure 4: The graph shows the win rate of the ϵ-greedy policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the green line the ϵ-greedy policy played first in all games against the Left policy with $\epsilon = 0.01$. For the blue line the ϵ-greedy policy played first in all games against the Left policy with $\epsilon = 0.05$. For the purple line the ϵ-greedy policy played first in all games against the Left policy with $\epsilon = 0.10$. For the red line the ϵ-greedy policy played first in all games against the Left policy with $\epsilon = 0.15$.

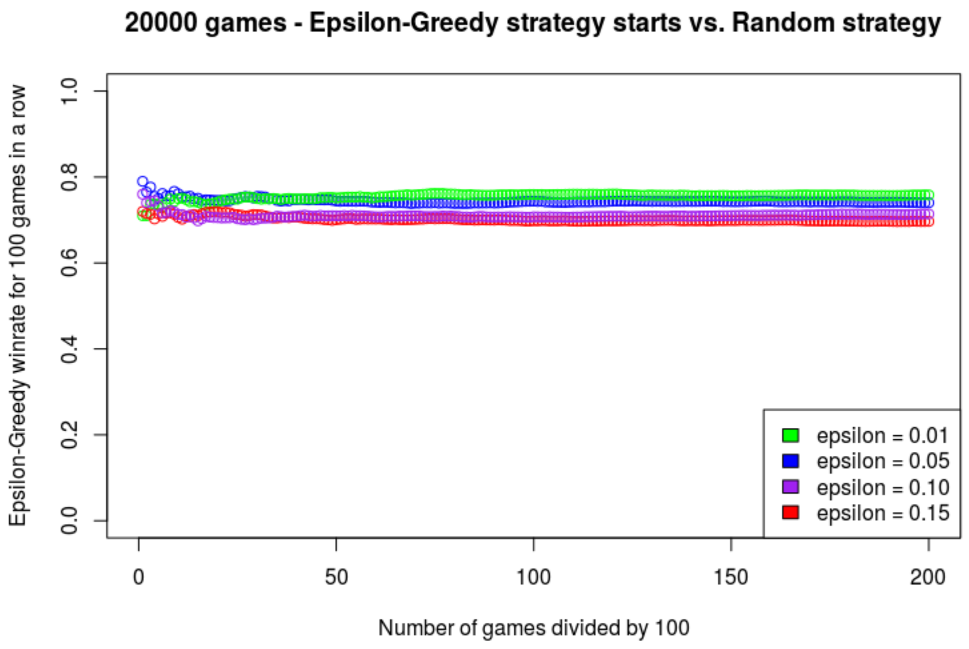


Figure 5: The graph shows the win rate of the ϵ-greedy policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the green line the ϵ-greedy policy played first in all games against the Random policy with $\epsilon = 0.01$. For the blue line the ϵ-greedy policy played first in all games against the Random policy with $\epsilon = 0.05$. For the purple line the ϵ-greedy policy played first in all games against the Random policy with $\epsilon = 0.10$. For the red line the ϵ-greedy policy played first in all games against the Random policy with $\epsilon = 0.15$.

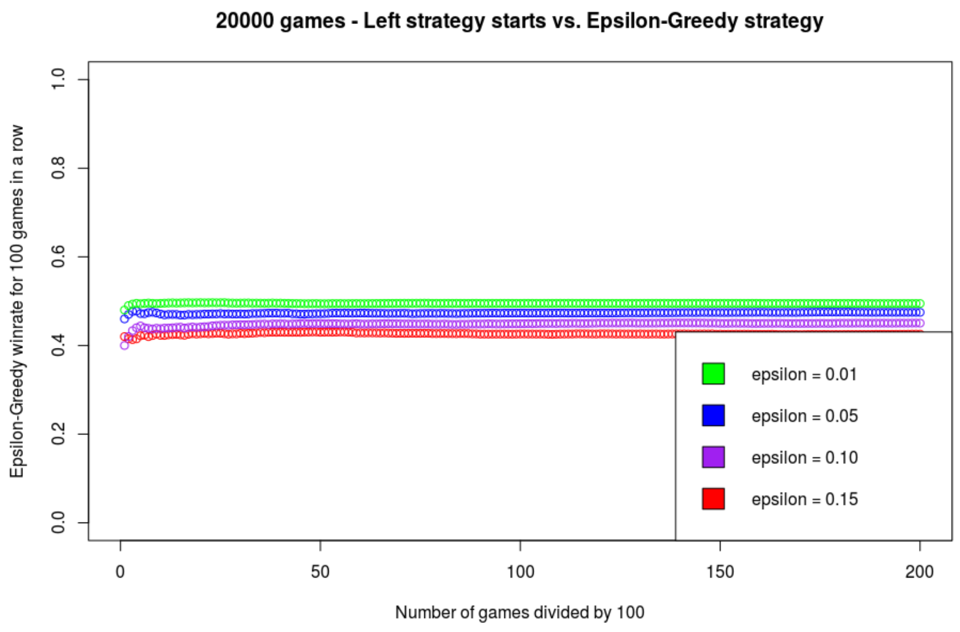


Figure 6: The graph shows the win rate of the ϵ-greedy policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the green line the ϵ-greedy policy played second in all games against the Left policy with $\epsilon = 0.01$. For the blue line the ϵ-greedy policy played second in all games against the Left policy with $\epsilon = 0.05$. For the purple line the ϵ-greedy policy played second in all games against the Left policy with $\epsilon = 0.10$. For the red line the ϵ-greedy policy played second in all games against the Left policy with $\epsilon = 0.15$.

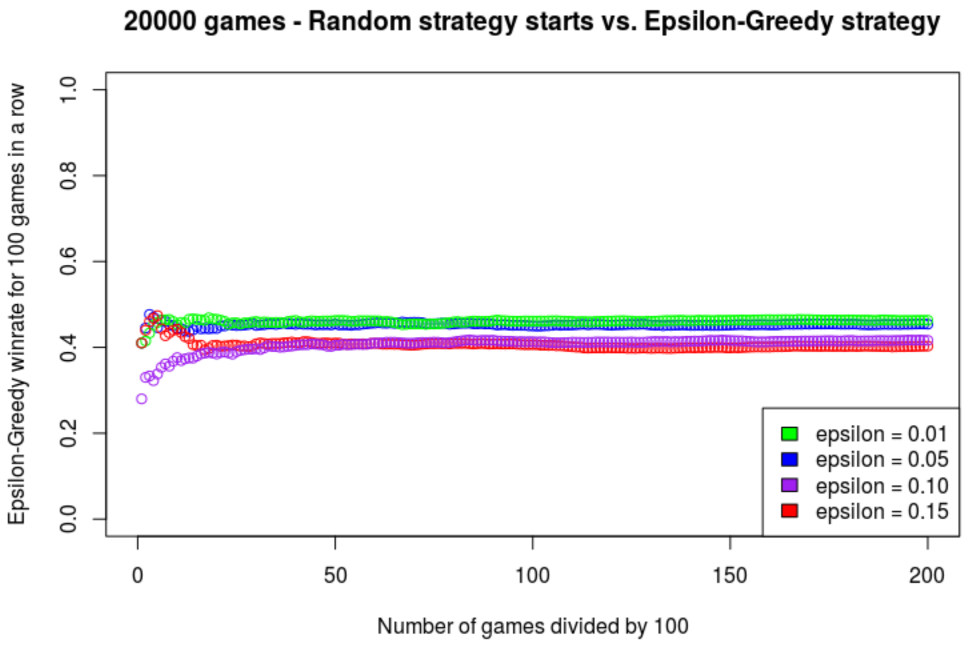


Figure 7: The graph shows the win rate of the ϵ-greedy policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the green line the ϵ-greedy policy played second in all games against the Random policy with $\epsilon = 0.01$. For the blue line the ϵ-greedy policy played second in all games against the Random policy with $\epsilon = 0.05$. For the purple line the ϵ-greedy policy played second in all games against the Random policy with $\epsilon = 0.10$. For the red line the ϵ-greedy policy played second in all games against the Random policy with $\epsilon = 0.15$.

In Figure 4 we see that the value of ϵ has an impact on the learning curve and the long-term win rate. The impact by the value of ϵ on the learning curve is unclear, based on the games displayed in Figure 5, 6, and 7. However, based on all figures we can see that a lower value of ϵ, generally results in a higher long-term win rate.

Theoretically, it would make sense that a lower value of ϵ, would result in a longer learning curve with higher long-term win rate. With a low value of ϵ, exploitation happens more frequently, and it takes longer to converge to an approximate optimal policy. It might make sense to choose a higher value of ϵ against some opponents though. However, since we know that the non-contextual ϵ-greedy policy is not suited for this reinforcement learning problem, I will not look further into this idea. From now on, I will use $\epsilon = 0.05$ when comparing the ϵ-greedy policy to other policies.

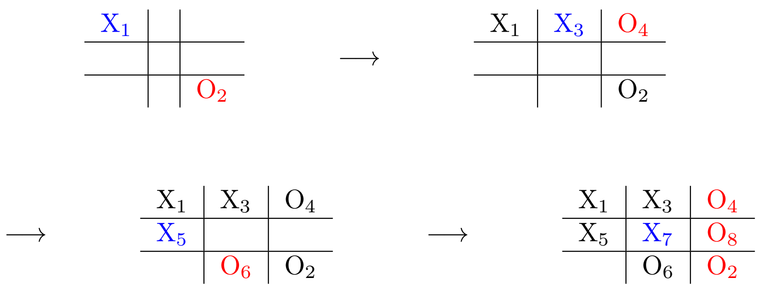
As we can see in Figure 6 and 7, the win rate stays below approximately 45-50% when playing as second. This is due to the fact that the non-contextual ϵ-greedy policy, is not suited for this kind of problem. The action-value $V(a)$ is represented as a single vector, where each element in the vector corresponds to the action-value of a. Below we have an example of the action-value vector, when playing against the Left player, where the Left player plays first:

```
[0,0,0,-0.200,0.038,-0.200,0.0475,-0.328,-0.200]
```

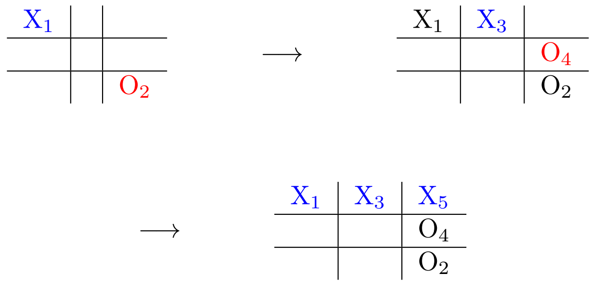
The action policy is to play three moves starting from the bottom-left, and going to the top-right. However, if the opponent plays in the middle column we will not change our policy. Instead, we will follow through with the rest of our "best" actions, even though we cannot possibly win with this policy. We will most likely lose, and losing will eventually change our "best" policy. Therefore, we will never converge towards an optimal policy.

When playing against the Left player, our reward vector $V(a)$, will always contain zeros for the first three actions. This corresponds to the top row on the game board. We cannot get a reward from the top row, since the Left player always plays the top row first. This will inhibit us from getting above 50 % win rate. We always choose the top-right action when having less than 2 positive action-values in the vector $V(a)$. However, if we have 2 or more positive action-values, we instantly lose, since we will not place a piece on the top row, before the Left policy wins.

Below we can see a visual illustration of how the ϵ-greedy policy plays versus the Left policy. The Left policy plays first using cross pieces and the ϵ-greedy policy plays second using circle pieces. For example, X_3 denotes a cross piece that was placed as the third piece on the game board by the Left policy.



We see that the ϵ-greedy algorithm wins. The algorithm now updates the action-value for the middle right action, such that $V(6)$ becomes positive. A new game is played, and the ϵ-greedy algorithm will follow a new policy.



We see that the ϵ-greedy algorithm now loses. The algorithm now updates the action-value for the middle-right action, such that $V(6)$ becomes negative. A new game is played, and the ϵ-greedy algorithm will follow something similar to the previous policy above. The policy will then win again using the top-right action. This constant win and lose pattern will continue since we do not take our opponents moves into account when playing.

2.2.3 Improvements to the ϵ-greedy policy

We need to use a contextual policy for the Tic-Tac-Toe problem. A policy will not be effective unless we can react to the opponents moves while playing.

2.3 Contextual ϵ-greedy policy

The contextual ϵ-greedy policy is similar to the non-contextual ϵ-greedy policy. The difference will be that our action-value V will depend on the state s however. This means that we will have a reward vector for each state s. Our decision policy will also depend on the state s. For this policy, we therefore have a most rewarding action $h(s)$ for each state s. The decision policy is a stochastic policy denoted by

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A_s|} & , \text{ if } h(s) = a \\ \frac{\epsilon}{|A_s|} & , \text{ otherwise} \end{cases},$$

where $|A_s|$ denotes the number of possible actions in state s.

Similarly to the non-contextual ϵ-greedy policy, we use ϵ to determine the probability of choosing a random action a. The action-value $V(a, s)$ for some action-state pair a, s is defined as the sum of rewards received as a result of transitioning to states $s'_{i,a,s}$. Here, $s'_{i,a,s}$ denotes the i'th state we visited after performing action a in state s at trial t. $V(a, s)$ is denoted by

$$V(a, s) = \alpha \sum_{i=1}^{T(a,s)} r(s'_{i,a,s}),$$

where $T(a, s)$ is a random variable that denotes the number of trials action a was performed in state s.

In this case we have that $\alpha = \frac{1}{5}$, since Tic-Tac-Toe is a non-stationary problem as explained earlier in section 2.2. The most rewarding action $h(s)$ is denoted by

$$h(s) = \text{argmax}_{a \in A_s} V(a, s).$$

2.3.1 Implementation details of the contextual ϵ-greedy policy

The reward function $r(s')$ for a state s' is denoted by

$$r(s') = \begin{cases} 1 & , \text{ state } s' \text{ is a won state} \\ 0 & , \text{ state } s' \text{ is an undecided state} \\ -1 & , \text{ state } s' \text{ which is a lost state} \end{cases}.$$

The algorithm implementation is similar to the non-contextual version of the ϵ-greedy algorithm. The pseudocode abstracts away from the implementation details and is heavily inspired by the pseudocode in Sutton's book [5, p. 24].

Algorithm 2 Contextual ϵ-greedy algorithm

```
1: Initialize:
2: for all $a$ from 1 to $|A|$ do
3:   for all $s$ from 1 to $|S|$ do
4:     $V(a, s) = 0$ //The action-value for some action in some state
5:     $N(a, s) = 0$ //The number of times some action was played in some state
6:   end for
7: end for
8: Loop:
9: while playing do
10:   $p$ = uniformly distributed random value between 0 and 1
11:   $Action$ = action from $A_s$, chosen uniformly at random
12:   if $p > \epsilon$ then
13:     $Action = \text{argmax}_{a \in A_s} V(a, s)$
14:   end if
15:   $Reward = Play(Action)$
16:   $N(a, s) = N(a, s) + 1$
17:   $V(a, s) = V(a, s) + \frac{1}{N(a,s)} \cdot (Reward - V(a, s))$
18: end while
```

Computational complexity

When I analyze the computational complexity I will be using asymptotic notation. The initialization process takes $O(|S| \cdot |A|)$ where $|S|$ is the total number of possible states and

$|A|$ is the total number of possible actions. Assume however that we initialize $V(a,s)$ and $N(a,s)$ in the `while` loop. This will mean that we only have to initialize all actions for a state, when we encounter the state for the first time. This will allow for a more general asymptotic running time of $O(|A| \cdot \min\{R, |S|\})$, where $|A|$ is the number of possible actions and R is the total rounds played by the algorithm. The `argmax` function will run in $O(|A|)$, and it will run for all rounds R that the algorithm plays. Thus resulting in a running time of $O(|A| \cdot R)$.

2.3.2 Performance of the contextual ϵ-greedy policy

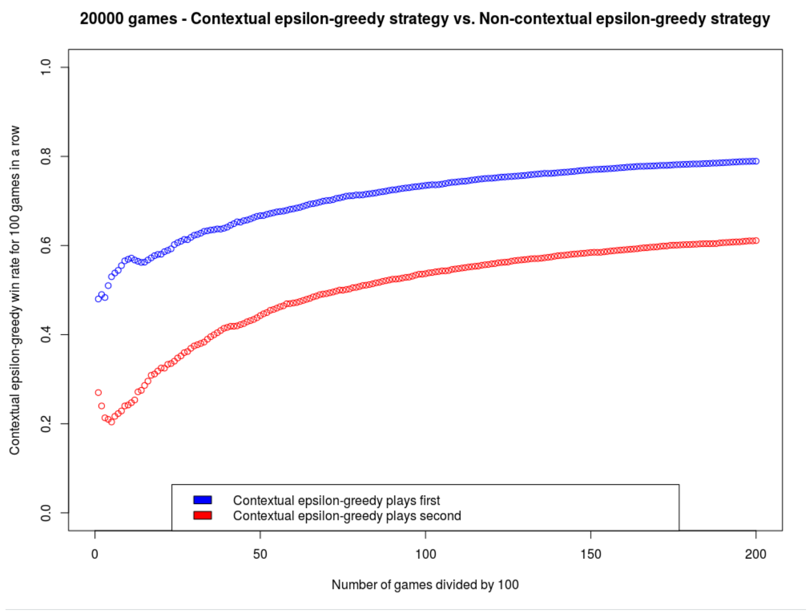


Figure 8: The graph shows the win rate of the contextual ϵ-greedy policy with $\epsilon = 0.05$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the blue line the contextual ϵ-greedy policy played first in all games against the non-contextual ϵ-greedy policy. For the red line the contextual ϵ-greedy policy played second in all games against the non-contextual ϵ-greedy policy.

In Figure 8 we see that the win rate peaks at 80% when the contextual policy plays the first move. When the contextual policy plays the second move, the policy peaks at 60% win rate. For the first few states we will always see the following action-value vector:

```
[0,0,0,0,0,0,0,0].
```

This means that we still lose in a lot of games, since the contextual ϵ-greedy policy chooses actions uniformly at random in the first few moves. If a state does not directly lead to a win or a loss, we will not learn anything, essentially playing randomly until reaching such a state.

2.3.3 Improvements to the contextual ϵ-greedy policy

We need a policy that learns what actions to choose in every state, and not just a state that directly results in a reward.

2.4 Contextual UCB policy

I will compare the performance of the contextual ϵ-greedy policy to the performance of the contextual UCB (upper confidence bound) policy. To determine what action to play, we calculate a confidence interval of the return of an action a for a given state s. We choose the action that has the highest upper confidence bound. The decision policy is a deterministic policy denoted by

$$\pi(s) = \text{argmax}_{a \in A_s} C(a, s),$$

where $C(a, s)$ is a function that calculates an upper confidence bound. $C(a, s)$ is denoted by

$$C(a, s) = V(a, s) + c \cdot \sqrt{\frac{\log(t)}{T(a, s)}},$$

where c is the constant that determines the confidence level and the square root term describes the uncertainty of action a [5, 28]. $V(a, s)$ has the same definition as for the contextual ϵ-greedy policy. $T(a, s)$ is a random variable that denotes the number of trials, where action a was chosen in state s. We now denote t by

$$t = \sum_{a,s} T(a, s).$$

2.4.1 Performance of the contextual UCB policy

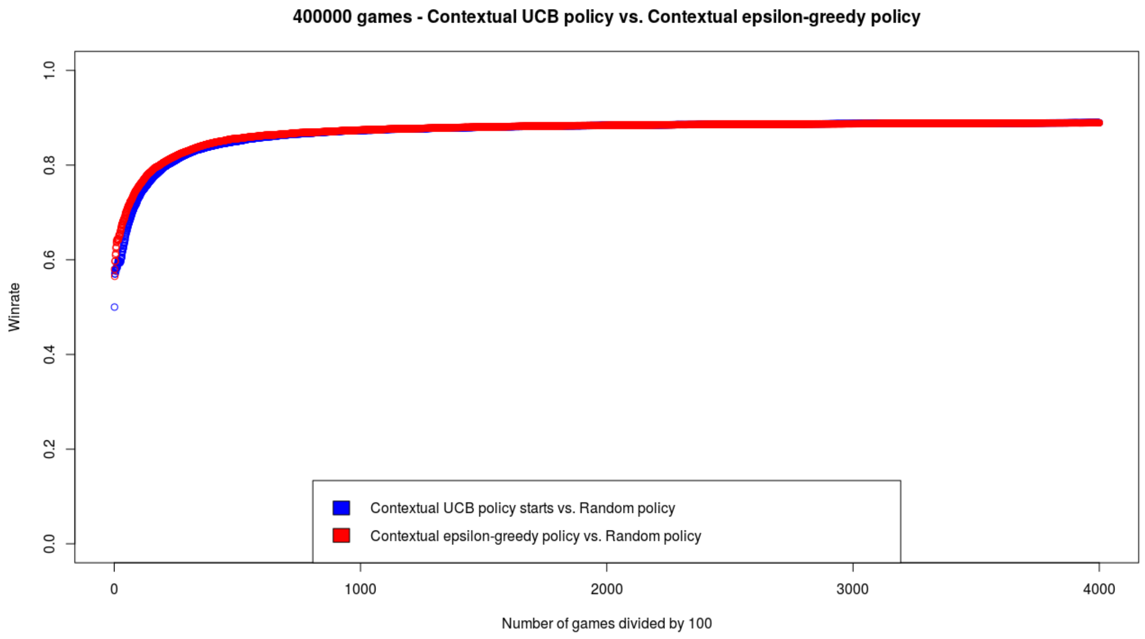


Figure 9: The graph shows the win rate of the contextual ϵ-greedy policy with $\epsilon = 0.01$ and the contextual UCB policy with $c = 0.01$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 40,000. Both the contextual ϵ-greedy policy and the contextual UCB policy played first against the Random policy. For the blue line the contextual UCB policy played first in all games against the Random policy. For the red line the contextual ϵ-greedy policy played first in all games against the Random policy.

In Figure 9 we see that the long-term win rate of the contextual UCB policy is just as high or slightly higher than it was for the contextual ϵ-greedy policy. The contextual UCB policy however has a slower and longer learning curve.

Neither policy is effective for the Tic-Tac-Toe problem, since both policies are choosing actions uniformly at random in all states, which cannot directly lead to a win or a loss. The contextual UCB policy is slightly superior judging by the long-term win rate when it is compared to the the contextual ϵ-greedy policy. This also seems to be the case in Sutton's and Barto's book with regards to the non-contextual policies [5, p. 28].

We might get a different result depending on the parameters of ϵ and c. It seems that a lower value of ϵ generally results in a better performance against the UCB policy however. When comparing the performance with parameters $\epsilon = 0.01$ and $c = 0.01$, the contextual UCB policy only had a lead on the win rate of approximate 0.022 percentage point after 400,000 games.

It is more complicated to apply the upper confidence bound method to other policies that are fit for non-stationary problems. Therefore I will avoid using the UCB policy onwards, based on the fact that the ϵ-greedy policy performs relatively well against the UCB policy [5, p. 28].

2.5 On-policy first-visit Monte Carlo control policy

For this policy we partition trials into episodes, such that an episode consists of multiple trials. This is to ensure a well-defined reward for non-stationary problems. Thus, we will avoid choosing actions uniformly at random in all states that do not lead to a win or loss. I will refer to this policy as the Monte Carlo ϵ-greedy policy.

We partition trials into episodes that must terminate at some point. Only after termination will the policy and action-value function update [5, p. 75]. An episode begins when the first move of a game is played and terminates when a player wins, loses or a draw happens. An action-value $V_{\pi_k}(a, s)$ is used to determine the reward of taking action a in state s under policy π_k. Using a first-visit policy means that we will set the action-value $V_{\pi_k}(a, s)$ for some action a and state s to be the average of the reward that follows the first time s is visited and action a is performed for all episodes [5, p. 79]. Under a Monte Carlo policy, an action-value function for action a in state s is defined as the the sum of rewards $r(s'_{i,a,s})$ for all final states $s'_{i,a,s}$ encountered in all episodes where action a was played in state s. Here $s'_{i,a,s}$ is the i'th final state we have encountered in some episode where we performed action a in state s. We only have a single final state per episode. $T_k(a, s)$ is a random variable denoting the number of times action a was performed in state s after k episodes. The action-value $V_{\pi_k}(a, s)$ is denoted by

$$V_{\pi_k}(a, s) = \frac{1}{T_k(a, s)} \sum_{i=1}^{T_k(a,s)} r(s'_{i,a,s}).$$

This action-value function applies to Tic-Tac-Toe since each reward can be determined by the final state in each episode. We refer to this policy as a first-visit Monte Carlo policy, since any state can only be visited once during an episode, and that the final state for any episode can determine the reward.
This Monte Carlo policy is meant to approximate an optimal policy π^*. We can approximate an optimal policy and optimal action-value since we utilize a technique called generalized policy iteration (GPI). With generalized policy iteration we maintain an action stochastic policy π_k and an action-value function V_{π_k} for the k'th episode. The action-value V_{π_k} is changed based on our policy π_k and then we find a new and improved policy π_{k+1} based on the action-value function. We can find a new and improved policy π_{k+1} if and only if we move towards a greedy policy with respect to the action-value. This means that we choose the action a to be our most rewarding action for policy π_k if the action maximizes $V_{\pi_k}(a, s)$ for a state s when constructing a policy π_{k+1} [5, p. 80]. However, we also allow suboptimal actions to be selected infinitely often since we need to explore all options in order to trust that we converge to the actual optimal policy [5,

p. 79]. For this implementation the policy will have an ϵ-greedy approach, meaning that we have a stochastic policy π_k where we choose the most rewarding action $h_{\pi_k}(s)$ most of the time, however we choose some random action a with probability ϵ for each trial. For $k \geq 0$ the policy is denoted by

$$\pi_k(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A_s|} & , \quad \text{if } h_{\pi_k}(s) = a \\ \frac{\epsilon}{|A_s|} & , \quad \text{else} \end{cases},$$

where $|A_s|$ denotes the number of possible actions in state s. As we can see, actions that are not necessarily most rewarding have a non-zero probability of being chosen, that is how we maintain exploration.

The most rewarding action $h_{\pi_k}(s)$ for $k \geq 1$, for policy π_k, and for state s is denoted by

$$h_{\pi_k}(s) = \text{argmax}_{a \in A_s} G_{\pi_{k-1}}(a, s).$$

For all s, the most rewarding action $h_{\pi_k}(s)$ for $k = 0$ is chosen uniformly at random from the set of possible actions A_s in state s.

2.5.1 Implementation details of the Monte Carlo ϵ-greedy policy

Here, the Tic-Tac-Toe reward given a state s' is denoted by:

$$r(s') = \begin{cases} 1 & , \quad \text{if the state } s' \text{ is a won state} \\ 0 & , \quad \text{if the state } s' \text{ is an undecided state} \\ -1 & , \quad \text{if the state } s' \text{ is a lost state} \end{cases}$$

The algorithm implementation partition trials into episodes such that an episode lasts from the start of a Tic-Tac-Toe game until a win, loss, or draw occurs. The pseudocode abstracts away from implementation details and heavily inspired by the pseudocode in Sutton's and Barto's book [5, p. 83]

Algorithm 3 On-policy first-visit MC control (Using ϵ-greedy policy)

```
Initialize:
for $a$ from 1 to $|A|$ do
  for $s$ from 1 to $|S|$ do
    $V(a,s) = 0$ //The action-value for some action in some state
    $Returns(a,s) =$ empty list
    $\pi(a|s) =$ optimal action chosen uniformly at random
  end for
end for
Loop:
while playing do
  Generate episode $e$ using $\pi$
  for all action-state pairs $a,s$ in episode $e$ do
    $Reward =$ the reward that follows the first occurrence of pair a,s
    Append $Reward$ to $Returns(a,s)$
    $V(a,s) =$ average of $Returns(a,s)$
  end for
  for all states $s$ in episode $e$ do
    $h_\pi(s) = \text{argmax}_{a \in A_s} V(a,s)$
    for all actions $a$ in episode $e$ do
      if $a == h_\pi(s)$ then
        $\pi(a|s) = 1 - \epsilon + \frac{\epsilon}{|A_s|}$
      else
        $\pi(a|s) = \frac{\epsilon}{|A_s|}$
      end if
    end for
  end for
end while
```

Computational complexity

When we analyze the computational complexity using asymptotic notation we see that the initialization process takes $O(|S| \cdot |A|)$ where $|S|$ is the total number of possible states in Tic-Tac-Toe and $|A|$ is the total number of possible actions. However, assume that we initialize $V(a,s)$ and $N(a,s)$ in the `while` loop. Then we only initialize all actions for a state when we encounter it the first time. This makes the running time of the initialization process have an upper bound of $O(|A| \cdot \min\{|S|, R\})$, where R is the total number of rounds played by the algorithm. Generating an episode means that we choose the optimal action or some other action depending on the value of ϵ, making the choice takes constant time $O(1)$, the choice has to be made $O(R)$ times. After generating an episode we set the action-value for each a,s pair in the episode to the average of the $Returns(a,s)$ list. Let us denote the total number of episodes by $|E|$. Calculating the average has a worst-case running time of $O(|E|)$, since we have a reward for each episode. We calculate the average $O(R)$ times giving a running time of $O(|E| \cdot R)$. Lastly, remember that the `argmax` function will run in $O(|A|)$, and it will run for all rounds R where the algorithm plays, resulting in $O(|A| \cdot R)$ for calculating `argmax`. Based on our analysis, this algorithm will have a running time of $O((|A| + |E|) \cdot R)$. We can, however, improve the running time by making the computation of an average reward iterative. Using an iterative approach the running time of the algorithm becomes $O(|A| \cdot R)$. Using an iterative approach the memory usage will be $O(|A| \cdot |S|)$ since we store a vector for all actions and for each state to make the action-value function. I implemented the algorithm using an iterative approach, thus, reducing the computational complexity. The pseudocode in section 2.7.1 or 2.3.1 shows how to compute the action-value function iteratively.

Property of the method

So far, we have used the action-value V to denote the value of an action. However, we would like to introduce a state-value function $Q_{\pi_k}(s)$ to be the value of a state s. Here, we closely follow the definition from Sutton's and Barto's book for the state-value $Q_{\pi_k}(s)$ for policy π_k and state s as:

$$Q_{\pi_k}(s) = \sum_{\forall a \in A_s} \pi_k(a|s) V_{\pi_k}(a, s)$$

Here $\pi_k(a|s)$ is the conditional probability of action a given a state s, which will be defined later in this section. [5, p. 83] We can use the state-value to evaluate a policy π_k for episode k against a policy π_{k+1} for episode $k+1$. If and only if for all states s and for all $k \geq 0$ the inequality $Q_{\pi_k}(s) \leq Q_{\pi_{k+1}}(s)$ holds, then we will be able to approximate an optimal policy π^* and an optimal state-value Q_{π^*}. This is called the policy improvement theorem [5, p. 63].

2.5.2 Performance of the Monte Carlo ϵ-greedy policy

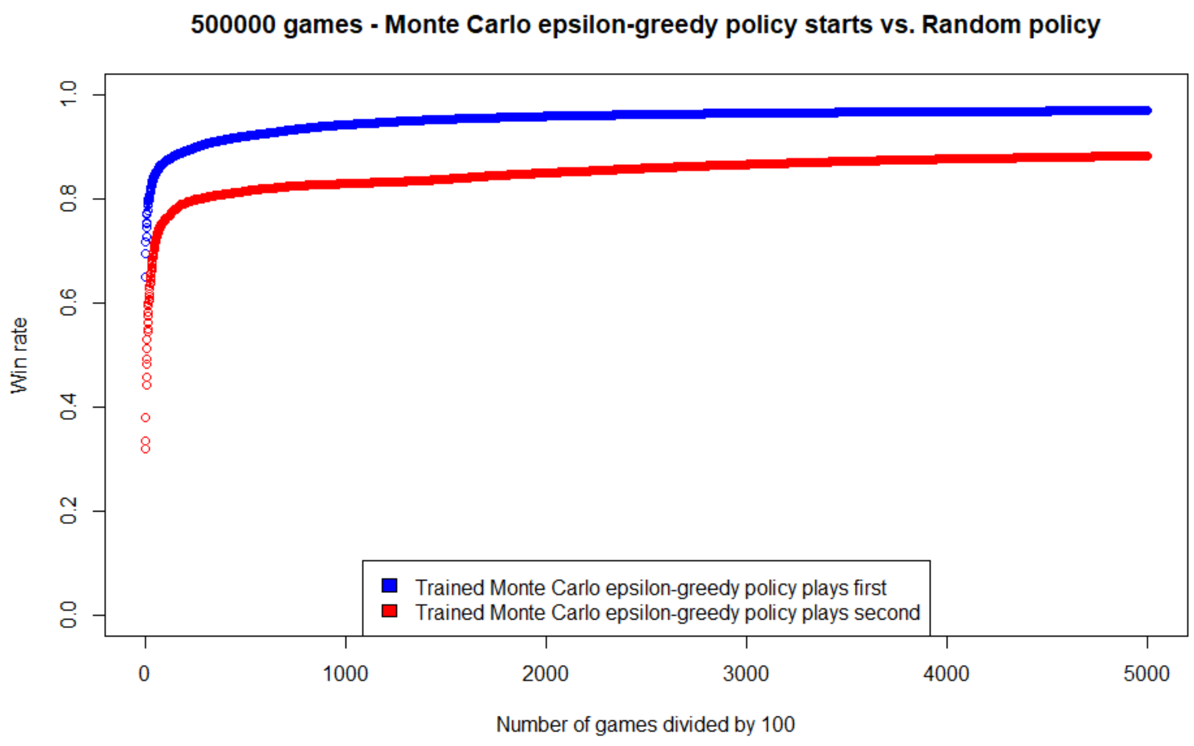


Figure 10: The graph shows the win rate of the Monte Carlo ϵ-greedy policy with $\epsilon = 0.01$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 500,000. For the blue line the on-policy first-visit Monte Carlo control policy played first in all games against the Random policy. For the red line the Monte Carlo ϵ-greedy policy played second in all games against the Random policy.

In Figure 10 we see that the blue line seems to converge at approximately 98 % win rate. For the blue line the Monte Carlo policy played first against the Random policy. The red line does not seem to have converged since it still has a noticeable positive trend going towards the 500,000'th game. For the red line the Monte Carlo policy played second against the Random policy. We can see that the Monte Carlo policy has a better win rate against the Random policy, compared to the previously mentioned policies. However, it would be interesting to see if the Monte Carlo policy converged to the optimal policy. In Figure 11 we used the action-value function generated from these 500,000 games to play 20,000 games with $\epsilon = 0$.

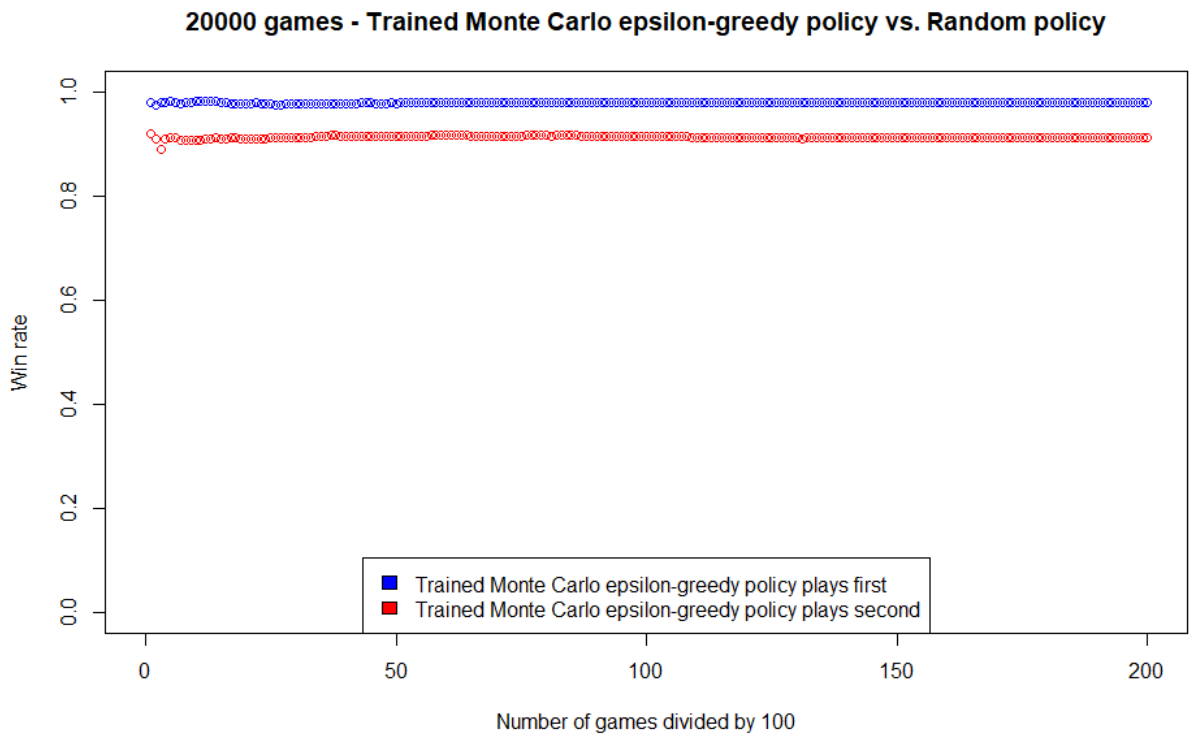


Figure 11: The graph shows the win rate of the trained Monte Carlo ϵ-greedy policy with $\epsilon = 0$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the blue line the trained on-policy first-visit Monte Carlo control policy played first in all games against the Random policy. For the red line the trained Monte Carlo ϵ-greedy policy played second in all games against the Random policy.

In Figure 11 we see that the Monte Carlo policy approached a win rate of approximately 98% when playing first against the Random policy. Out of 20,000 games the Monte Carlo policy lost 0 games and had 379 draws when playing first. According to this test the Monte Carlo policy has an approximately 2% chance of getting a draw against the Random policy when playing first. When playing first, the Monte Carlo policy seems to have almost converged to the optimal strategy mentioned in section 1.1.2 since the optimal number of draws is 0,05%. However, we can look at a few examples of the action-value function and compare this to the actual optimal Tic-Tac-Toe strategy mentioned in section 1.1.2. Below we see examples of Tic-Tac-Toe game boards where each empty field displays a Monte Carlo policy action-value. The Monte Carlo policy plays as "X" and the Random policy plays as "O". The values below have been rounded to the nearest 3 decimals.

0.618	0.610	0.953
0.607	0.772	0.580
0.701	0.538	0.745

$\longrightarrow$

0.986	0.644	X_1
0.612	0.747	O_2
0.774	0.818	0.741

$\longrightarrow$

X_3	O_4	X_1
0.771	0.867	O_2
0.971	0.601	0.421

$\longrightarrow$

X_3	O_4	X_1
0.968	O_6	O_2
X_5	0.600	-1.000

$\longrightarrow$

X_3	O_4	X_1
X_7	O_6	O_2
X_5		

When comparing the example above to the optimal strategy mentioned in section 1.1.2, we do not play the exact same optimal Tic-Tac-Toe strategy. However, when taking a closer look at the actions of the Monte Carlo policy, we see that the Monte Carlo policy actually perform similar actions to those of the optimal strategy in section 1.1.2. According to the action-value function the most optimal first action X_1 is the top-right. This is similar to the strategy in section 1.1.2, the Monte Carlo policy simply plays in a different corner. It does not matter which corner the Monte Carlo policy plays first since the board is symmetrical. The actions X_3, X_5, X_7 are different from what the optimal strategy in section 1.1.2 performs. However, the actions played by the Monte Carlo policy are still just as optimal as the actions played in section 1.1.2, since the actions performed by the Monte Carlo policy lead to a guaranteed win, by making more than one winning position available at once.

In Figure 11 we see that the Monte Carlo policy approached a win rate of approximately 91,4% when playing second against the Random policy. Out of 20,000 games the Monte Carlo policy lost 441 games and had 1,308 draws when playing second. The policy did not become optimal since it should have lost 0 games and should have had approximately 1680 draws. However, it seems as if the policy got close to being optimal. We can compare a few examples of the action-state function to see if it resembles the optimal strategy presented in section 1.1.2. The Monte Carlo policy plays second as "O" (circle), and the Random strategy plays first as "X" (cross). The values below have been rounded to the nearest 3 decimals.

0.215	-0.273	0.155
-0.111	X_1	0.241
0.533	-0.064	0.474

$\longrightarrow$

0.643	0.091	X_3
0.100	X_1	0.000
O_2	0.240	0.429

$\longrightarrow$

O_4	-0.471	X_3
0.655	X_1	0.337
O_2	-0.000	X_5

$\longrightarrow$

O_4		X_3
O_6	X_1	
O_2		X_5

In the example above, the Monte Carlo policy wins and perform actions similar to those in the optimal Tic-Tac-Toe strategy mentioned in section 1.1.2. The actions that are performed are not exactly the same as in section 1.1.2, however they might be rotated or symmetrical. According to the strategy in section 1.1.2, the first move of the Monte Carlo policy O_2 should have been performed in the top-left corner. However, it does not make a difference as long as we choose a different corner and we rotate all other consecutive actions as well. Actions O_4, O_6 are also both similar to the optimal Tic-Tac-Toe strategy in section 1.1.2. The only difference is that the actions performed by the Monte Carlo policy are rotated accordingly to O_2.

2.6 Stochastic gradient ascent policy using inverse probability weighted estimator

This policy is described in Jonas Peter's project description - "Learning how to play Tic Tac Toe" [3, p. 10][1][1]. I will not implement the policy myself, since the implementation has been handed out to me in an R script. I will present this policy as it is presented in [3, p. 10], however, I will modify some of the notation to make it more similar to the rest of the notation used in this thesis. The purpose for this thesis, among other purposes, is to compare this inverse probability weighted (IPW) stochastic gradient ascent policy to the other policies implemented in this thesis. I will refer to this policy as the IPW gradient ascent policy. I use the phrase "an episode" as a synonym for "a game", meaning that an episode starts when the policy performs the first move and terminates when a win, loss, or draw happens. We determine an improved policy by maximizing the expected reward under the current policy π with respect to a preference of actions. We denote the stochastic decision policy π as the conditional probability of choosing action a given a state s.

$$\pi(a|s) = \mathbb{P}(A = a|S = s) = \frac{\exp(H(a,s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))}, \tag{5}$$

where $H(a,s)$ is defined as the numerical preference of choosing action a given state s. In the denominator of (5) we sum over all possible actions a' from the set of actions A_s given state s. We will use an inverse probability weighted estimator $\hat{\mathbb{E}}(\pi)$ to estimate the performance of a policy π. Assume that we know fixed probabilities $\pi_{data}(a_{k,t}|s_{k,t})$ that have been used to generate data. Here, $a_{k,t}$ denotes the action performed by the policy in state $s_{k,t}$ for the t'th move in episode k. We denote the inverse probability estimator $\hat{\mathbb{E}}(\pi)$ by

$$\hat{\mathbb{E}}(\pi) = \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})},$$

where $|E|$ denotes the total number of episodes and $T(k)$ denotes the number of trials in episode k. For each episode k we have a score (reward) $r_k(s'_k)$ for the final state s'_k in episode k. Here, we weigh each reward by the ratio between the current policy π and the probability of the generated data π_{data} (the old policy). The weight will then be larger for a reward if the policy π makes the reward more likely than the previous policy π_{data} did. To find a new improved policy we will use gradient ascent to estimate an optimal preference $H(a,s)$ that will maximize our inverse probability weighted estimator. To accomplish this, we update the preference $H(a,s)$ for the action $a \in A_s$ that was performed in state s by

$$H(a,s) \leftarrow H(a,s) + \lambda \cdot \frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(a,s)}.$$

We update the preference $H(\tilde{a},s)$ for the actions $\tilde{a} \in A_s/\{a\}$ that were not performed in state s by

$$H(\tilde{a},s) \leftarrow H(\tilde{a},s) + \lambda \cdot \frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(\tilde{a},s)},$$

where the constant λ denotes the step size. The partial derivatives are given by

$$\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(a,s)} = \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \left(1 - \frac{\exp(H(a,s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))}\right) \cdot \frac{\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})},$$

$$\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(\tilde{a},s)} = \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{-\exp(H(\tilde{a},s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))} \cdot \frac{\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})}.$$

[1]The page number might change, however, the description can be found in Appendix A - "Inverse probability weighting"

The derivation of $\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(a,s)}$ and $\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(\tilde{a},s)}$ are not shown in [3, p. 10]. However, I decided to derive both and include the derivations in this thesis to give the reader an idea of how the partial derivatives are derived.

$$\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(a,s)} = \frac{\partial}{\partial H(a,s)} \left(\frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})} \right)$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\left(\frac{\partial}{\partial H(a,s)} \left(\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t}) \right) \right)}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\left(\frac{\partial}{\partial H(a,s)} \left(\frac{\exp(H(a,s))}{\exp(H(a,s)) + \sum\limits_{a' \in A_s, a' \neq a} \exp(H(a',s))} \right) \right) \cdot \prod_{t=1, a_{k,t} \neq a, s_{k,t} \neq s}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\left(\frac{-\exp(H(a,s)) \cdot \exp(H(a,s))}{\left(\exp(H(a,s)) + \sum\limits_{a' \in A_s, a' \neq a} \exp(H(a',s)) \right)^2} + \frac{\exp(H(a,s))}{\exp(H(a,s)) + \sum\limits_{a' \in A_s, a' \neq a} \exp(H(a',s))} \right)}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})} \cdot \prod_{t=1, a_{k,t} \neq a, s_{k,t} \neq s}^{T(k)} \pi(a_{k,t}|s_{k,t})$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\left(1 - \frac{\exp(H(a,s))}{\exp(H(a,s)) + \sum\limits_{a' \in A_s, a' \neq a} \exp(H(a',s))} \right) \frac{\exp(H(a,s))}{\exp(H(a,s)) + \sum\limits_{a' \in A_s, a' \neq a} \exp(H(a',s))}}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})} \cdot \prod_{t=1, a_{k,t} \neq a, s_{k,t} \neq s}^{T(k)} \pi(a_{k,t}|s_{k,t})$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \frac{\left(1 - \frac{\exp(H(a,s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))} \right) \frac{\exp(H(a,s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))} \cdot \prod_{t=1, a_{k,t} \neq a, s_{k,t} \neq s}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|} \sum_{k=1}^{|E|} r_k(s'_k) \cdot \left(1 - \frac{\exp(H(a,s))}{\sum\limits_{a' \in A_s} \exp(H(a',s))} \right) \cdot \frac{\prod_{t=1}^{T(k)} \pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)} \pi_{data}(a_{k,t}|s_{k,t})}$$

$$\frac{\partial \hat{\mathbb{E}}(\pi)}{\partial H(\tilde{a},s)} = \frac{\partial}{\partial H(\tilde{a},s)}\left(\frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\prod_{t=1}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}\right)$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\left(\frac{\partial}{\partial H(\tilde{a},s)}\left(\prod_{t=1}^{T(k)}\pi(a_{k,t}|s_{k,t})\right)\right)}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\left(\frac{\partial}{\partial H(\tilde{a},s)}\left(\frac{\exp(H(a,s))}{\exp(H(\tilde{a},s))+\sum\limits_{a'\in A_s,a'\neq\tilde{a}}\exp(H(a',s))}\right)\right)\cdot\prod_{t=1,a_{k,t}\neq a,s_{k,t}\neq s}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\frac{-\exp(H(a,s))\cdot\exp(H(\tilde{a},s))}{\left(\exp(H(\tilde{a},s))+\sum\limits_{a'\in A_s,a'\neq\tilde{a}}\exp(H(a',s))\right)^2}\cdot\prod_{t=1,a_{k,t}\neq a,s_{k,t}\neq s}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\frac{-\exp(H(a,s))\cdot\exp(H(\tilde{a},s))}{\left(\sum\limits_{a'\in A_s}\exp(H(a',s))\right)^2}\cdot\prod_{t=1,a_{k,t}\neq a,s_{k,t}\neq s}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{\frac{-\exp(H(\tilde{a},s))}{\sum\limits_{a'\in A_s}\exp(H(a',s))}\cdot\frac{\exp(H(a,s))}{\sum\limits_{a'\in A_s}\exp(H(a',s))}\cdot\prod_{t=1,a_{k,t}\neq a,s_{k,t}\neq s}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

$$= \frac{1}{|E|}\sum_{k=1}^{|E|} r_k(s'_k)\cdot\frac{-\exp(H(\tilde{a},s))}{\sum\limits_{a'\in A_s}\exp(H(a',s))}\cdot\frac{\prod_{t=1}^{T(k)}\pi(a_{k,t}|s_{k,t})}{\prod_{t=1}^{T(k)}\pi_{data}(a_{k,t}|s_{k,t})}$$

2.6.1 Implementation details of the IPW gradient ascent policy

The code for the implementation of the IPW gradient ascent policy was handed to me, therefore I will not put much focus on the implementation details of this policy. However, one important thing to mention is the computational complexity. I will describe the computational complexity of the algorithm for this policy using asymptotic notation. If the policy is implemented exactly as described in the previous section, the algorithm will be much slower than the other policies implemented in this thesis. When calculating the partial derivatives we calculate the product of all $\pi(a_{k,t})$ for all episodes k and all trials t. This makes the running time of each time we calculate the partial derivative $(|E|\cdot R)$, where $|E|$ is the total number of episodes and R is the total number of rounds played by the algorithm. If we calculate the partial derivatives each time an episodes terminates we will get a total running time of $O(|E|^2\cdot R)$. There are options to reduce the computational complexity such as using an iterative approach when calculating the partial derivatives. However, the implementation that I was given does not use an iterative approach. Instead, a more practical approach have been used to make the algorithm run faster. The total number of episodes is limited to 500. This means that once for every 500'th episode, data from the last 500 episodes such as rewards, actions, or states will be erased. Before the data is erased, the partial derivatives will be calculated and the preference $H(a,s)$ will be updated.

2.6.2 Performance of the IPW gradient ascent policy

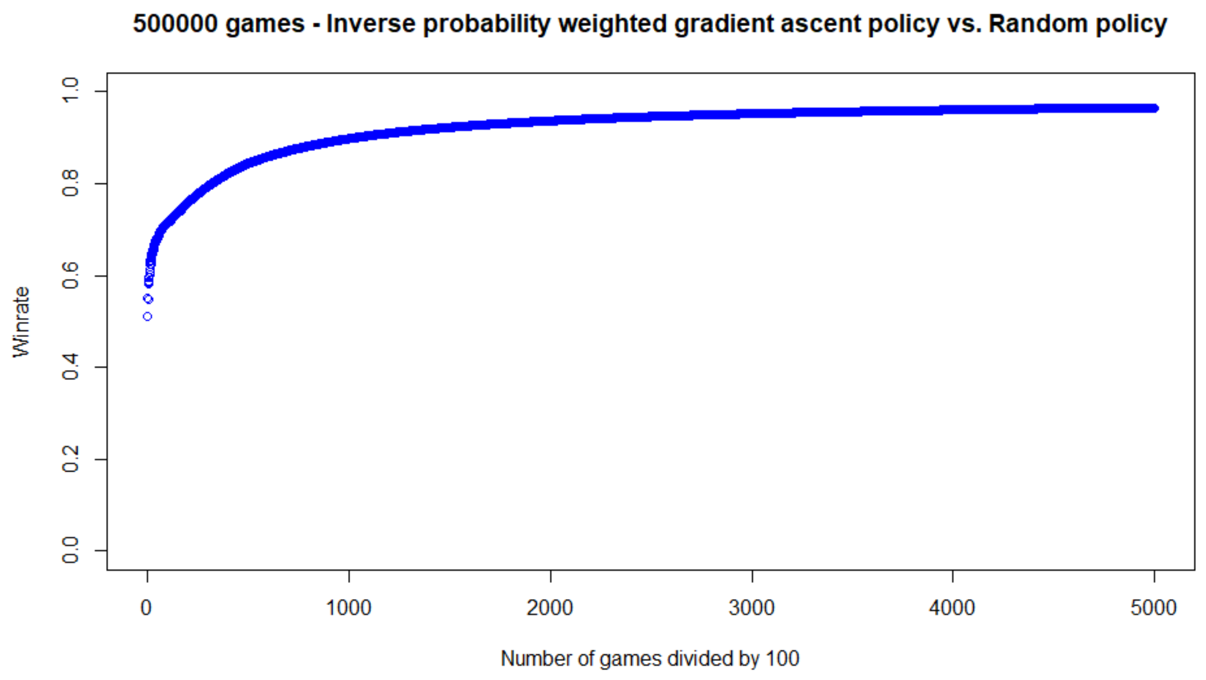


Figure 12: The graph shows the win rate of the IPW gradient ascent policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 500,000. For the blue line the IPW gradient ascent policy played first in all games against the Random policy.

In Figure 12 the IPW gradient ascent policy played 500,000 games against the Random policy. The policy learns slower than the Monte Carlo policy mentioned in section 2.5, in the sense that the learning curve has a lower increase for each game played. This could be due to the fact that the policy only improves once for every 500'th game, and that data is erased afterwards. Another factor that could impact the learning curve would be the value of λ, the implementation has set $\lambda = \frac{20}{|E|}$. However, a value of $\lambda = \frac{20}{|E|}$ might not be as effective compared to using a constant, for example $\lambda = 0.1$, since the Tic-Tac-Toe problem is non-stationary.

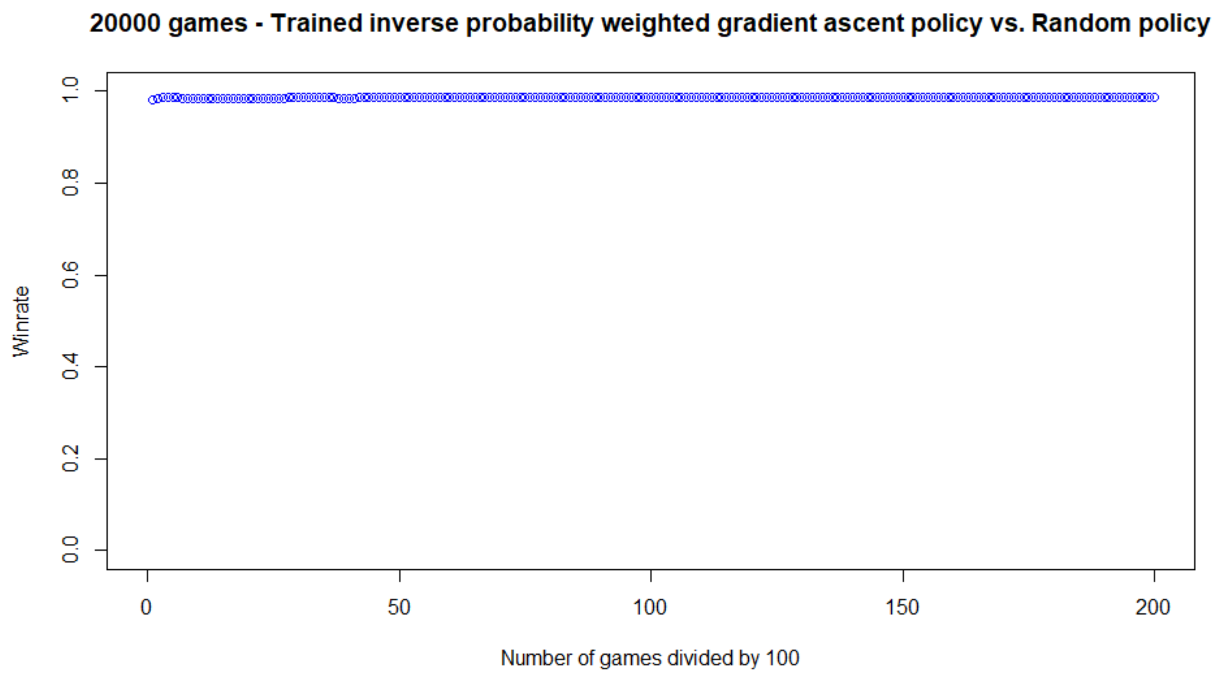


Figure 13: The graph shows the win rate of the trained IPW gradient ascent policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the blue line the trained IPW gradient ascent policy played first in all games against the Random policy.

After training the IPW gradient ascent policy on 500,000 games against the Random policy where the IPW gradient policy played first, the trained policy played an extra 20,000 games against the Random policy where the IPW gradient ascent policy also played first. Out of 20,000 games the policy lost 29 games and had 241 draws. The policy had an average win rate of approximately 98,6% as shown in Figure 13. The win rate should have been approximately 99,5% if the policy had approximated the optimal strategy in section 1.1.2. Even though the policy might have approximated a strategy that cannot lose, it might still choose suboptimal actions since all actions have a non-zero probability of being chosen. This could be the reason why the IPW gradient ascent policy did not lose 0 games. Compared to the trained Monte Carlo policy in section 2.5 which lost 0 games with $\epsilon = 0$, the IPW gradient ascent policy had fewer draws. The reason why the trained Monte Carlo policy in section 2.5 lost 0 games could be due to the fact that suboptimal actions have a 0% probability of being chosen when $\epsilon = 0$.

2.7 On-policy first-visit Monte Carlo stochastic gradient ascent

In this bachelor thesis I will refer to this policy as the Monte Carlo gradient ascent policy. This policy is based on a mixture of the two policies: the non-contextual stochastic gradient ascent policy from Sutton's and Barto's book [5, p. 28] and the on-policy first-visit Monte Carlo policy, also from Sutton's and Barto's book [5, p. 83]. The idea of mixing the two policies was my own.

Changes had to be made however, as the two policies could not be applied directly together. A Monte Carlo policy separates the average of rewards between each action a that is performed in state s into an action-value function $V_{\pi_k}(a, s)$. This makes it possible for us to approximate the value of an action [5, p. 47]. However, the non-contextual stochastic gradient ascent policy uses a cumulative gain $G(t)$ to update the preference $H_t(a_t)$ for an action a_t that was performed at trial t, similar to what the IPW gradient ascent policy does in section 2.6. The cumulative gain is the sum of rewards r_{a_i} for all trials i where $0 < i \leq t$.

A non-contextual iterative approach for updating the preference H using stochastic gradient ascent is defined in Sutton's and Barto's book [5, p. 29]. The update $H_{t+1}(a_t)$ for the action a_t that was performed at trial t is denoted by

$$H_{t+1}(a_t) = H_t(a_t) + \alpha(r_{a_t} - \frac{1}{t} \cdot G(t))(1 - \pi_t(a_t))$$

where α is the step size. The update $H_{t+1}(\tilde{a}_t)$ for the actions $\tilde{a}_t$ that were not performed at trial t is denoted by

$$H_{t+1}(\tilde{a}_t) = H_t(\tilde{a}_t) + \alpha(r_{a_t} - \frac{1}{t} \cdot G(t))(-\pi_t(\tilde{a}_t))$$

where α denotes the step size. According to Sutton and Barto, the average of the cumulative gain $\frac{1}{t} \cdot G(t)$ is used as a baseline for the reward r_{a_t} [5, p. 29]. If the reward is larger than the baseline, then the probability of performing the action a_t is increased. If the reward is smaller than the baseline, then the probability of performing the action a_t is decreased. The decision policy is a soft-max distribution denoted by

$$\pi(a) = \frac{\exp(H(a))}{\sum_{a' \in A} \exp(H(a'))}$$

where A is the set of actions [5, p. 28].

As mentioned before, the non-contextual gradient ascent policy is not suited for non-stationary contextual problems. We can however construct a Monte Carlo stochastic gradient ascent (MC-SGA) policy by making a few changes. As mentioned in section 2.5, we partition trials into episodes in order to ensure a well-defined return [5, p. 75]. Each episode begins at the first move of a game and terminates when a player wins, loses, or a draw occurs. We define a preference $H_{\pi_k}(a, s)$ for action a in state s using policy π_k.

Now that we have a contextual preference, we also have a contextual stochastic decision policy $\pi_k(a|s)$ defined as the conditional probability of choosing action a given state s, during episode k. The stochastic decision policy $\pi_k(a|s)$ is also a soft-max distribution. The policy is denoted by

$$\pi_k(a|s) = \frac{\exp(H_{\pi_k}(a, s))}{\sum\limits_{a' \in A_s} \exp(H_{\pi_k}(a', s))} \tag{6}$$

where A_s is the set of possible actions given state s. Since we want to implement a Monte Carlo policy, we must also define an action-value function. We will use the definition in section 2.5. At the end of this section I will present an iterative formula however.

We are ready to consider how we change the method of updating preference $H_{\pi_k}(a, s)$. Instead of using the average cumulative gain $\frac{1}{t} \cdot G(t)$ at trial t, we instead use an action-value function. For example, when updating an action $\tilde{a}$ that was not performed in state

s during episode k, we will replace $\frac{1}{t} \cdot G(t)$ with $V_{\pi_k}(\tilde{a}, s)$.

When the baseline is an action-value function instead of an average cumulative gain, the gradient policy will learn faster. Imagine that some action a was performed in state s during episode k. The action-value $V_{\pi_k}(a, s)$ is close to 0 and the reward is a win, therefore the update to the preference $H_{\pi_{k+1}}(a, s)$ will be relatively large. Consider now the same situation where we use an average of the cumulative gain instead. The average of the cumulative gain will be close to 1 in this case, therefore the reward counts as almost nothing, thereby making the update to preference $H_{\pi_{k+1}}(a, s)$ relatively small.

This implies that the learning will be faster when using an action-value function. We update the preference $H_{\pi_{k+1}}(a, s)$ for actions a that was performed in state s during episode k by

$$H_{\pi_{k+1}}(a, s) = H_{\pi_k}(a, s) + \alpha(r(s'_{T_k(a,s),a,s}) - V_{\pi_k}(a, s))(1 - \pi_k(a|s))$$

where α is the step size. We update the preference $H_{\pi_{k+1}}(\tilde{a}, s)$ for actions $\tilde{a}$ that was not performed in state s during episode k by

$$H_{\pi_{k+1}}(\tilde{a}, s) = H_{\pi_k}(\tilde{a}, s) + \alpha(r(s'_{T_k(a,s),a,s}) - V_{\pi_k}(\tilde{a}, s))(-\pi_k(\tilde{a}|s)),$$

where α is the step size. We set the initial value $H_{\pi_0}(a, s) = 0$ for all a and s.

Since $T_k(a, s)$ denotes the number of times action a was performed in state s after k episodes, we can denote the most recent reward as $r(s'_{T_k(a,s),a,s})$ for the most recent final state $s'_{T_k(a,s),a,s}$. The fact that we compute the partial derivative for the performed action a, means that we use an on-policy method. If it was an off-policy method, we would compute the partial derivative using the greedy action. The greedy action is the action with the highest action-value. We denote the iterative action-value $V_{\pi_{k+1}}(a, s)$ for action a in state s by

$$V_{\pi_{k+1}}(a, s) = V_{\pi_k}(a, s) + \frac{1}{T_k(a, s) + 1} \cdot (r(s'_{T_k(a,s)+1,a,s}) - V_{\pi_k}(a, s))$$

where $V_{\pi_0}(a, s) = 0$ for all a and s.

2.7.1 Implementation details of the Monte Carlo gradient ascent policy

The Tic-Tac-Toe reward given a state s' is here denoted by:

$$r(s') = \begin{cases} 1 & , \quad \text{if the state } s' \text{ is a won state} \\ 0 & , \quad \text{if the state } s' \text{ is an undecided state} \\ -1 & , \quad \text{if the state } s' \text{ is a lost state} \end{cases}$$

The algorithm implementation partition trials into episodes such that an episode lasts from the start of a Tic-Tac-Toe game until a win, loss, or draw occurs. The pseudocode abstracts away from the implementation details and is heavily inspired by the pseudocode in Sutton's and Barto's book [5, p. 83].

Algorithm 4 On-policy first-visit MC stochastic gradient ascent (MC-SGA) policy

```
1: Initialize:
2: for $a$ from 1 to $|A|$ do
3:   for $s$ from 1 to $|S|$ do
4:     $V(a, s) = 0$ //The action-value for some action in some state
5:     $N(a, s) = 0$ //The number of times action $a$ was performed in state $s$
6:     $H(a, s) = 0$ //Each action have equal preference in the beginning
7:   end for
8: end for
9: Loop:
10: while playing do
11:   Generate episode $e$ using $\pi$
12:   for all action-state pairs $a, s$ in episode $e$ do
13:     $N(a, s) = N(a, s) + 1$
14:     $Reward$ = the reward that follows the first occurrence of pair s,a
15:     $V(a, s) = V(a, s) + \frac{1}{N(a,s)} \cdot (Reward - V(a, s))$
16:     $H(a, s) = H(a, s) + \alpha \cdot (Reward - V(a, s)) \cdot (1 - \pi(a|s))$
17:     for all actions $\tilde{a} \neq a$ in $A_s$ do
18:       $H(\tilde{a}, s) = H(\tilde{a}, s) + \alpha \cdot (Reward - V(\tilde{a}, s)) \cdot (-\pi(\tilde{a}|s))$
19:     end for
20:   end for
21: end while
```

Computational complexity

When I analyze the computational complexity I will be using asymptotic notation. As mentioned previously the initialization process can be done iteratively in the `while`-loop, thus resulting in a running time of $O(|A| \cdot \min |S|, R)$, where $|A|$ is the total number of actions, $|S|$ is the total number of states and R is the total number of rounds played by the algorithm. The algorithm then generates an episode by selecting which actions are to be played. Selecting an action can be done in constant time $O(1)$, we have to select an action for each round the algorithm plays however, resulting in a running time of $O(R)$. All action-state pairs a, s in episode e are visited afterwards using a loop. This loop will run $O(R)$ times, since we have one action-state pair a, s for each round that was played. For each action-state pair we will run a loop for all actions not performed $\tilde{a} \in A_s$. The running time of this loop is $O(|A|)$. The running time of this algorithm is therefore $O(R \cdot |A|)$.

2.7.2 Performance of the Monte Carlo gradient ascent policy

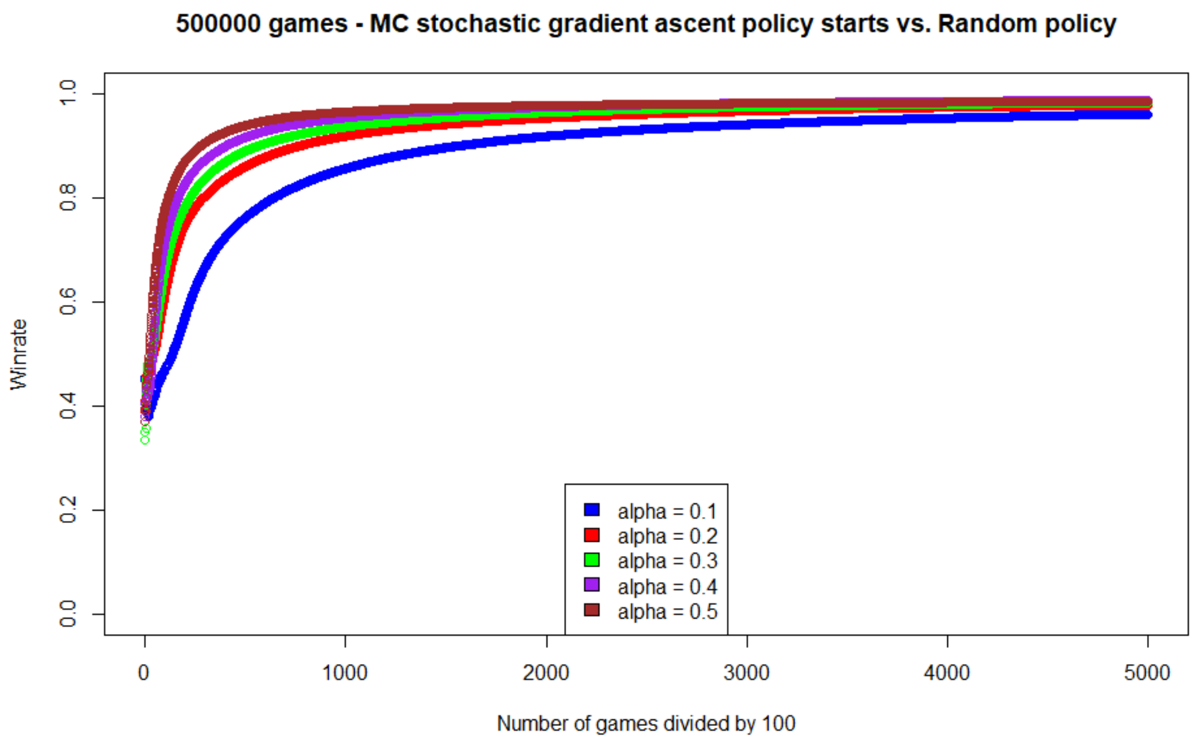


Figure 14: The graph shows the win rate of the Monte Carlo gradient ascent policy when playing first. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 500,000. For the blue line the Monte Carlo gradient ascent policy policy played first in all games against the Random policy.

In Figure 14 we see that the Monte Carlo gradient ascent policy learns faster than the inverse probability weighted gradient ascent policy when it plays first. The Monte Carlo ϵ-greedy policy learns faster than both when it plays first however. The optimal policy for learning fast has $\alpha = 0.5$. On the long term however, $\alpha = 0.4$ is slightly superior.

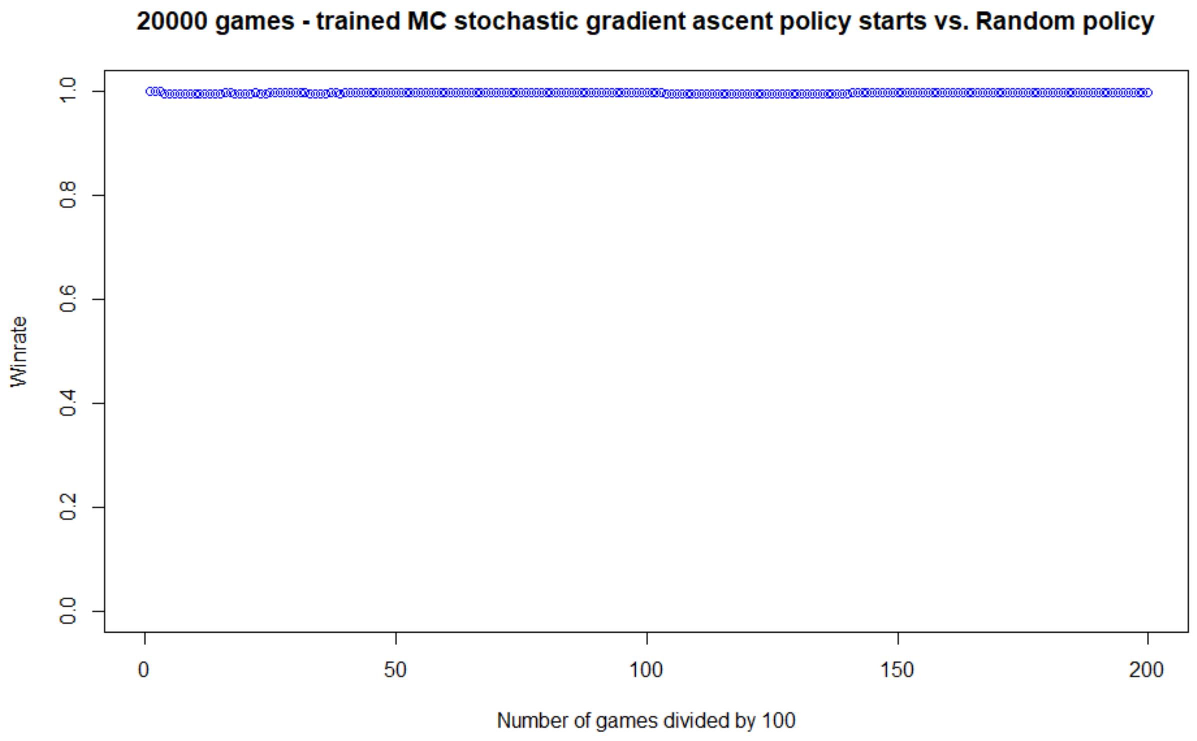


Figure 15: The graph shows the win rate of the trained Monte Carlo gradient ascent policy with $\alpha = 0.0$ when playing first. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 20,000. For the blue line the trained Monte Carlo gradient ascent policy policy played first in all games against the Random policy.

In Figure 15 the trained Monte Carlo gradient ascent policy played against the random strategy. Out of 20,000 games the Monte Carlo gradient ascent policy lost 2 games and had 58 draws. The Monte Carlo gradient ascent policy should have been able to get 0 losses using the optimal strategy presented in section 1.1.2. However, this could be due to the fact that suboptimal actions have a non-zero probability of being selected. As mentioned earlier, the trained Monte Carlo ϵ-greedy policy in section 2.5 played with $\epsilon = 0$, meaning that suboptimal actions had 0% chance of being selected. The Monte Carlo gradient ascent policy however, had far fewer draws than the Monte Carlo ϵ-greedy policy presented in section 2.5. Overall, when the Monte Carlo gradient ascent policy played 20,000 games after being trained, the policy had an average win rate of 99,6%. This means that the Monte Carlo gradient ascent policy approximated the optimal strategy presented in section 1.1.2 closely.

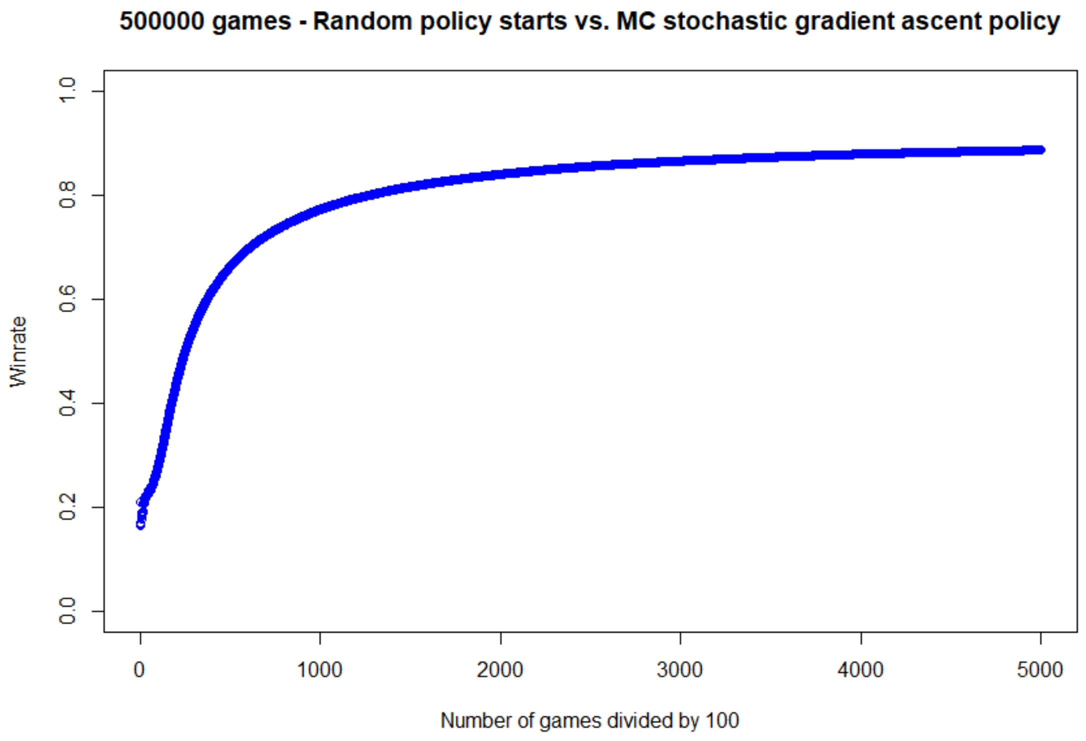


Figure 16: The graph shows the win rate of the Monte Carlo gradient ascent policy when playing second. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 500,000. For the blue line the Monte Carlo gradient ascent policy played second in all games against the Random policy.

In Figure 16 we see that the Monte Carlo gradient ascent policy learns slower than the Monte Carlo ϵ-greedy policy. This could be due to the fact that exploration is based on a preference. However, the learning does not flatten out as quickly, such that the Monte Carlo gradient ascent policy achieves a better long-term win rate. The fact that exploration is based on a preference also means that we are more likely to explore actions that will eventually lead to an approximated optimal strategy. When we continuously explore options with higher preference, in contrast to choosing uniformly at random, we achieve a more steady and faster learning. If we choose some action uniformly at random we will more often explore actions that are not a part of an optimal strategy.

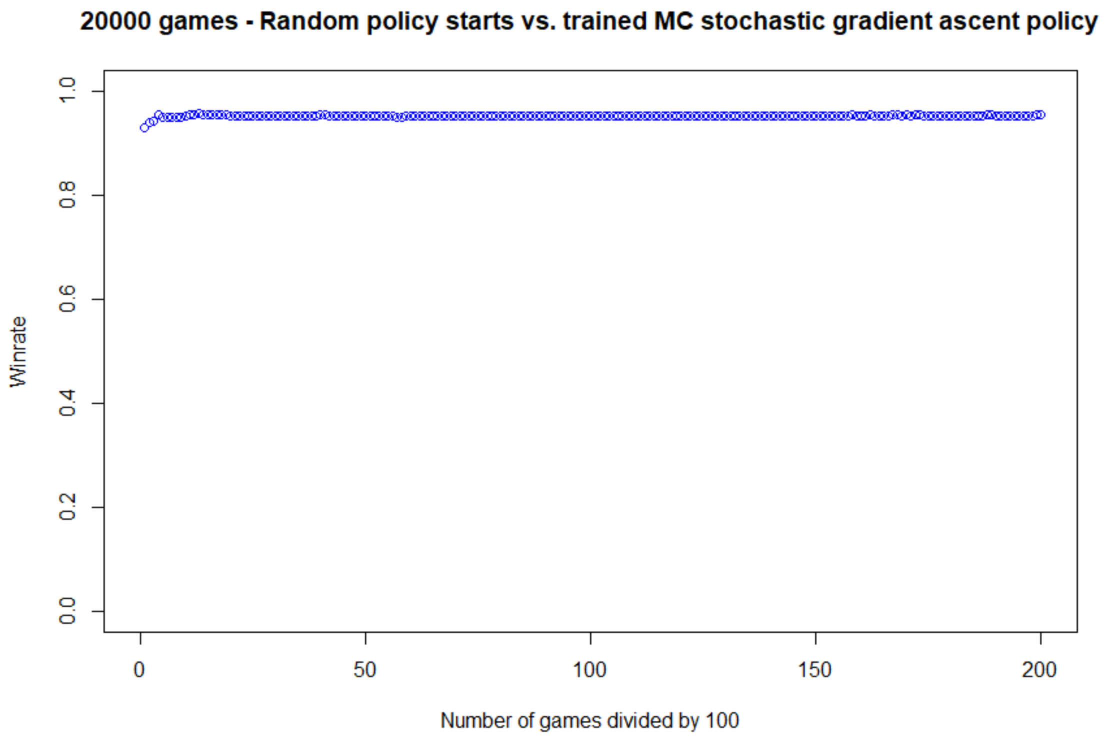


Figure 17: The graph shows the win rate of the trained Monte Carlo gradient ascent policy when playing second. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 500,000. For the blue line the trained Monte Carlo gradient ascent policy played second in all games against the Random policy.

In Figure 17 we see that the Monte Carlo gradient ascent policy achieves an average win rate of 0.95%. Surprisingly, this is better than what is possible using the optimal strategy in section 1.1.2. According to the optimal strategy in section 1.1.2, we should expect a maximum average win rate of approximately 92,6%, since 8,4% percent of games should end in a draw. We should also expect 0% losses. However, the Monte Carlo gradient ascent policy lost 354 games and had 562 draws. It seems that the policy managed to get an even higher win rate by losing some games. Perhaps, since the opponent is playing uniformly at random a more greedy approach is actually better than the optimal strategy in section 1.1.2. Here, a greedy approach is to be understood as sacrificing some games in order to generally perform better. If the Monte Carlo gradient ascent policy had played the optimal strategy in section 1.1.2, we would have expected approximately 1680 draws and 0 losses. However, the cumulative gain is higher by losing 354 games and having 562 draws. In essence, the purpose of a reinforcement learning algorithm is to maximize the cumulative gain, as mentioned in section 1.2.

3 Experiments

3.1 Tournament

3.1.1 Start bracket

The first experiment will be a tournament where the decision policies (excluding Left policy and Random policy) will face off against each other. Below you can see the initial brackets for the tournament.

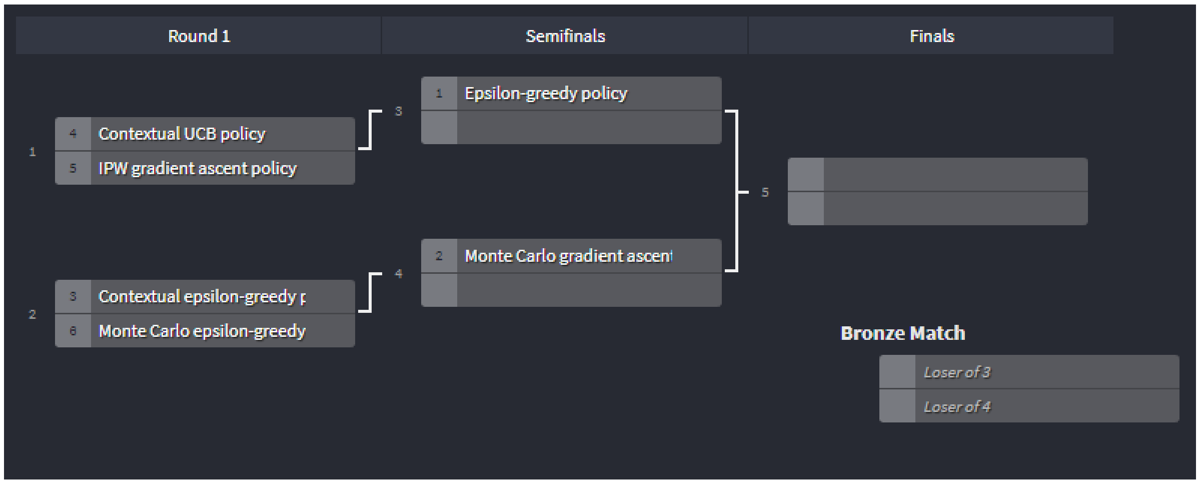

Round 1
Semifinals
Finals
1
4 Contextual UCB policy
5 IPW gradient ascent policy
2
3 Contextual epsilon-greedy p
6 Monte Carlo epsilon-greedy
3
1 Epsilon-greedy policy
4
2 Monte Carlo gradient ascent
5
Bronze Match
Loser of 3
Loser of 4

Figure 18: The initial brackets for the tournament where the non-trivial decision policies (excluding Left policy and Random policy) will face off against each other

In Figure 18 we see that the contextual UCB policy will face off against the IPW gradient ascent policy in match 1. In match 2, the contextual ϵ-greedy policy will face off against the Monte Carlo ϵ-greedy policy. The winner of match 1 will continue to match 3 in the semi-finals to face off against the ϵ-greedy policy. The winner of match 2 will continue to match 4 in the semi-finals to face off against the Monte Carlo stochastic gradient ascent policy. The winners of matches 3 and 4 will face off against each other in the finals. The two losers will face off in the bronze match for the second and third place in the tournament.

3.1.2 Rules

When a match begins both competing policies will be untrained. Each match will be carried out such that both policies get to play first and second against each other. The winner will have won while playing both first and second. If one policy wins as playing first but loses as playing second, we will choose the policy that has achieved the highest cumulative gain with respect to both starting positions. For example, if a policy wins as playing first, then its sum of all wins, losses, and draws must be higher than the sum of all wins, losses, and draws of the second policy. Lets assume that 10 games are played where one policy played first in 5 games and achieved 3 wins, 1 loss and 1 draw, this results in a cumulative gain of 2. The other policy played first in the remaining 5 games and achieved 3 wins and 2 draws then this results in a cumulative gain of 3, hence this other policy wins the match as playing first. We also have to decide who won the match as playing second. In this case the other policy also won as playing second since it also had a larger cumulative sum when playing second.

The number of games played for each match will be 160,000 games. When 80,000 games have been played the policies swap starting positions such that another 80,000 games can be played.

3.1.3 Expected outcome

I expect that the winner of match 1 will be the IPW gradient ascent policy since the contextual UCB policy does not learn from states that does not directly lead to a win, loss or draw. This means that the contextual UCB policy will choose actions uniformly at random most of the rounds.

I expect that the winner of match 2 will be the Monte Carlo ϵ-greedy, for the same reason that I think the contextual UCB policy will lose match 1.

I expect that the winner of match 3 will be the IPW gradient ascent policy since it will play against the non-contextual ϵ-greedy policy. The non-contextual ϵ-greedy policy will have a bad performance against the IPW gradient ascent policy since the context matters a great deal in Tic-Tac-Toe.

I expect that the winner of match 4 will be the Monte Carlo gradient ascent policy, since it generally performed better versus the Random strategy compared to how the Monte Carlo ϵ-greedy policy performed.

I expect that the winner of the finale will be the Monte Carlo gradient ascent policy, since it generally performed better versus the Random strategy compared to how the IPW gradient ascent policy performed.

I expect that the winner of the bronze match will be Monte Carlo ϵ-greedy policy, since it has performed better than the IPW gradient ascent policy against the Random policy, in terms of learning speed.

3.1.4 Match 1 - IPW gradient ascent policy vs. Contextual UCB policy

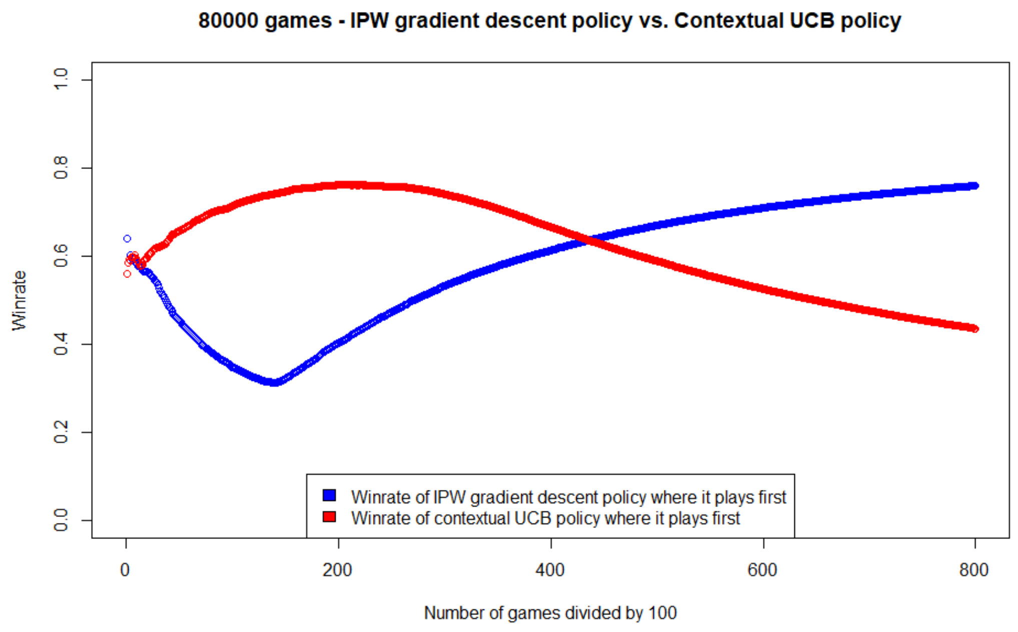


Figure 19: The win rate of the IPW gradient ascent policy and the Contextual UCB policy with $\alpha = \frac{1}{5}$ and $c = 0.1$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the IPW gradient ascent policy played first in all games against the contextual UCB policy. For the red line the contextual UCB policy played first in all games against the IPW gradient ascent policy.

Surprisingly, the contextual UCB policy performs quite well against the IPW gradient ascent policy as shown in Figure 19. The contextual UCB policy seems to have the upper hand since it is able to learn faster than the IPW gradient ascent policy. On the long-term it seems as though the IPW gradient ascent policy has the advantage. If the number of games played was higher, then it seems that the IPW gradient ascent would win. However, if the number of games played had been fewer the IPW gradient ascent might have lost.

Match result: IPW gradient ascent wins!

IPW gradient ascent				
	Wins	Losses	Draws	Cumulative gain
Plays first	60812	9849	9339	50963
Plays second	6793	34834	38373	-28041

Contextual UCB policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	34834	6793	38373	28041
Plays second	9849	60812	9339	-50963

3.1.5 Match 2 - Monte Carlo ϵ-greedy policy vs. Contextual ϵ-greedy policy

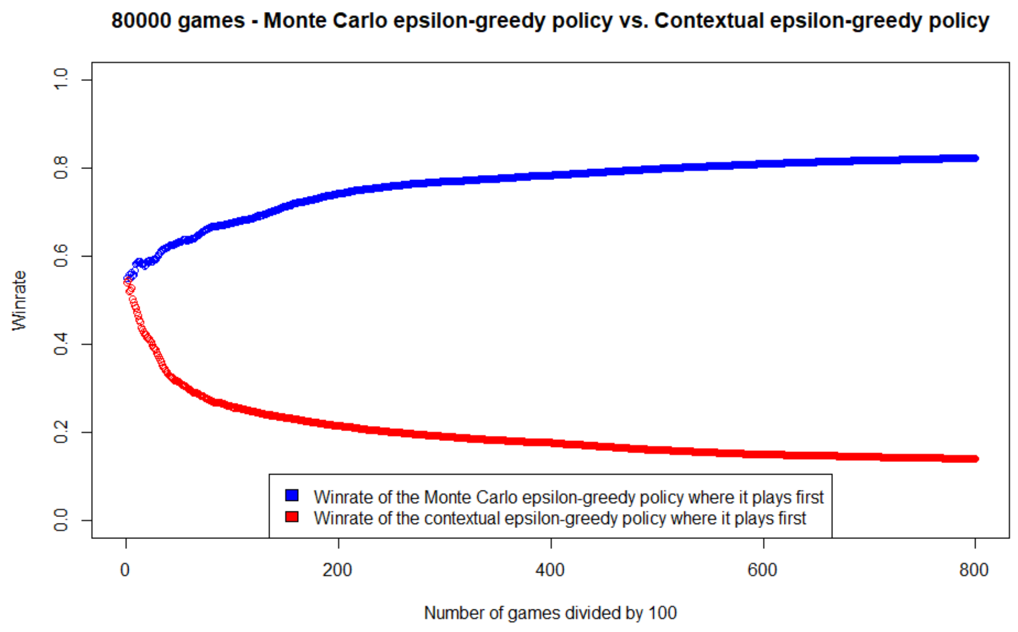


Figure 20: The win rate of the Monte Carlo ϵ-greedy policy with $\epsilon = 0.05$ and the contextual ϵ-greedy policy with $\epsilon = 0.05$ and $\alpha = \frac{1}{5}$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the Monte Carlo ϵ-greedy policy played first in all games against the contextual ϵ-greedy policy. For the red line the contextual ϵ-greedy policy played first in all games against the Monte Carlo ϵ-greedy policy.

It seems that the Monte Carlo ϵ-greedy policy is superior in terms of learning rate and long-term performance compared to the contextual ϵ-greedy policy. The number of games played should not make a difference on the outcome of the match.

Match result: Monte Carlo ϵ-greedy policy wins!

Monte Carlo ϵ-greedy policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	65805	2910	11285	62895
Plays second	19098	11114	49788	7984

Contextual ϵ-greedy policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	11114	19098	49788	-7984
Plays second	2910	65805	11285	-62895

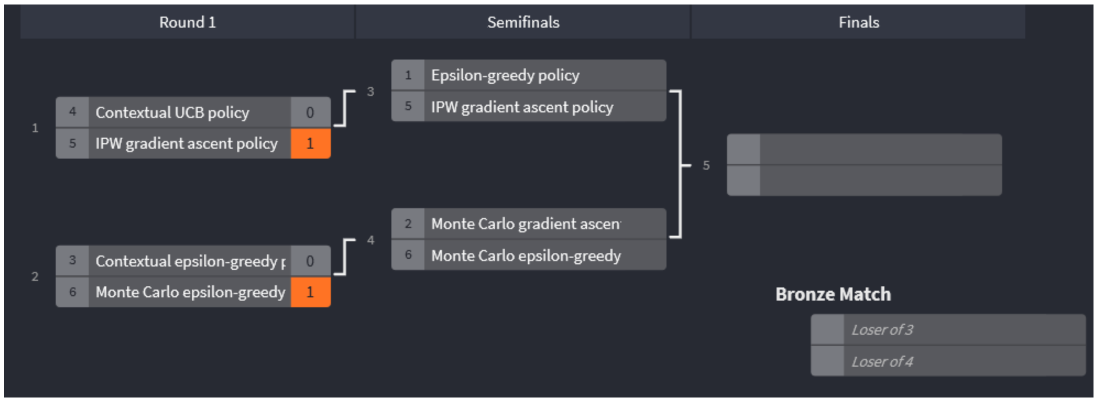

Round 1
Semifinals
Finals
1
4 Contextual UCB policy 0
5 IPW gradient ascent policy 1
2
3 Contextual epsilon-greedy p 0
6 Monte Carlo epsilon-greedy 1
3
1 Epsilon-greedy policy
5 IPW gradient ascent policy
4
2 Monte Carlo gradient ascen
6 Monte Carlo epsilon-greedy
5
Bronze Match
Loser of 3
Loser of 4

Figure 21: Match 1 and 2 have concluded so we are now in the semi-finals. Currently, all predictions have come true. However, the IPW gradient ascent policy performed worse than I expected against the contextual UCB policy. The IPW gradient ascent policy might not have made it to match 3 if fewer games had been played for each match.

3.1.6 Match 3 - IPW gradient ascent policy vs. Non-contextual ϵ-greedy policy

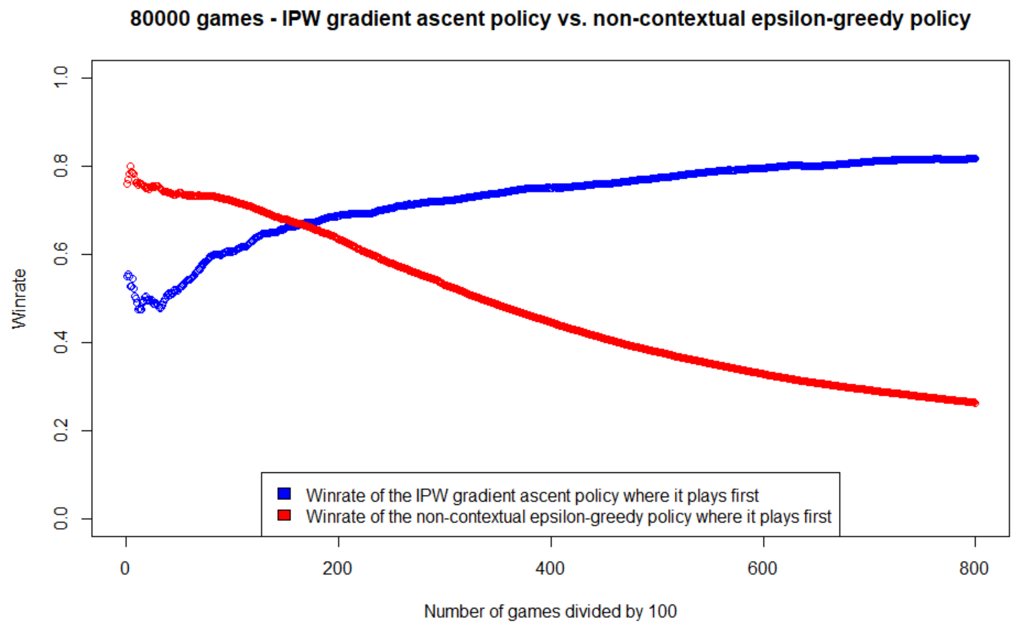


Figure 22: The win rate of the IPW gradient ascent policy and the non-contextual ϵ-greedy policy with $\epsilon = 0.05$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the IPW gradient ascent policy played first in all games against the non-contextual ϵ-greedy policy. For the red line the non-contextual ϵ-greedy policy played first in all games against the IPW gradient ascent policy.

Again, the IPW gradient ascent policy struggles in the early games. It seems that the ϵ-greedy policy has a chance of winning if the number of played games was lower. However, on the long-term the IPW gradient ascent policy performs better, this means that the more games we play, the more likely it is for the IPW gradient ascent policy to win against the non-contextual ϵ-greedy policy. However, it is surprising that the non-contextual policy can perform well against a contextual policy, as long as the learning rate of the contextual policy is slow enough.

Match result: IPW gradient ascent policy wins!

IPW gradient ascent policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	65378	4138	10484	61240
Plays second	42599	21041	16360	21558

Non-contextual ϵ-greedy policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	21041	42599	16360	-21558
Plays second	4138	65378	10484	-61240

3.1.7 Match 4 - Monte Carlo gradient ascent policy vs. Monte Carlo ϵ-greedy policy

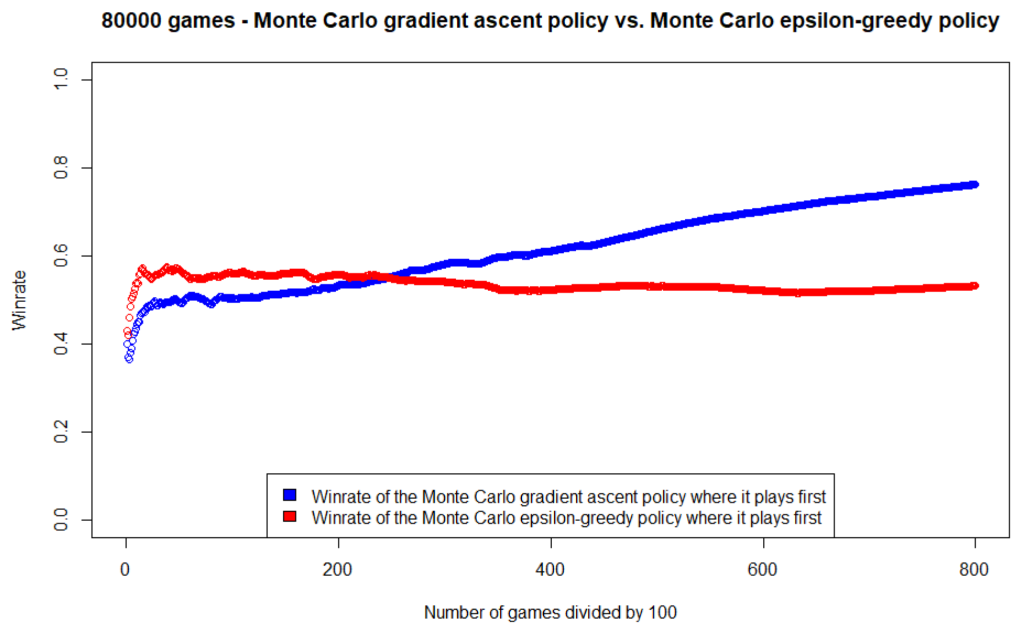


Figure 23: The win rate of the Monte Carlo gradient ascent policy with $\alpha = 0.01$ and the Monte Carlo ϵ-greedy policy with $\epsilon = 0.05$. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the Monte Carlo gradient ascent policy played first in all games against the Monte Carlo ϵ-greedy policy. For the red line the Monte Carlo ϵ-greedy policy played first in all games against the Monte Carlo gradient ascent policy.

The Monte Carlo ϵ-greedy policy seems to be superior with regards to learning at first, however, as more games are played the learning rate of the Monte Carlo gradient ascent policy never seems to flatten out. The Monte Carlo gradient ascent policy played with $\alpha = 0.01$, which is not the optimal value of α as shown in Figure 23. Therefore it could be that the initial learning rate of the Monte Carlo gradient ascent would have been superior with a different α. However, if changing the value of α does not make a difference, the Monte Carlo gradient ascent policy might have lost, if the games played was fewer than approximately 20,000. However, as the number of games increase, the Monte Carlo gradient descent policy definitely seems superior.

Match result: Monte Carlo gradient ascent policy wins!

Monte Carlo gradient ascent policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	61006	6032	12962	54974
Plays second	25066	42521	12413	-17455

Monte Carlo ϵ-greedy policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	42521	25066	12413	17455
Plays second	6032	61006	12962	-54974

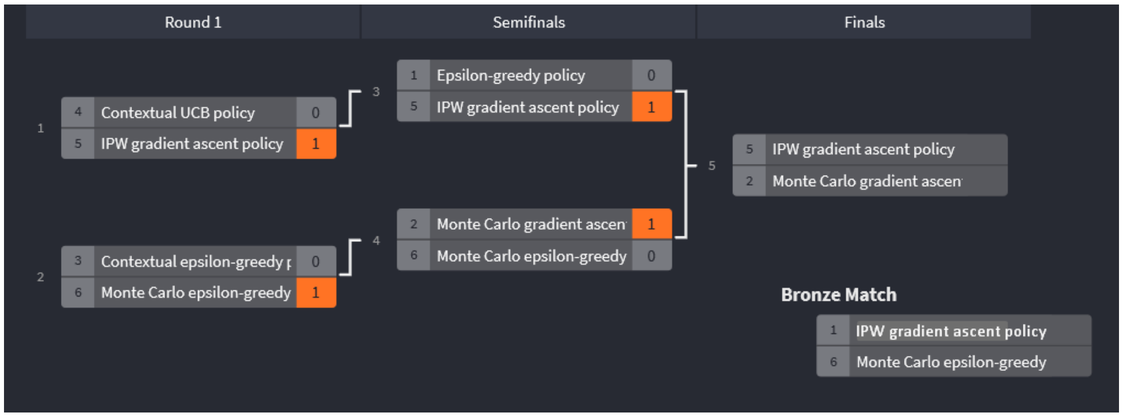

Round 1
Semifinals
Finals
1
4 Contextual UCB policy 0
5 IPW gradient ascent policy 1
2
3 Contextual epsilon-greedy p 0
6 Monte Carlo epsilon-greedy 1
3
1 Epsilon-greedy policy 0
5 IPW gradient ascent policy 1
4
2 Monte Carlo gradient ascen 1
6 Monte Carlo epsilon-greedy 0
5
5 IPW gradient ascent policy
2 Monte Carlo gradient ascen
Bronze Match
1 IPW gradient ascent policy
6 Monte Carlo epsilon-greedy

Figure 24: Match 3 and 4 have concluded so we are now in the finals. Currently, all predictions have come true. After the final the bronze match will be played. I decided to let the Monte Carlo ϵ-greedy policy face off against the IPW gradient policy in the bronze match, since the Monte Carlo ϵ-greedy generally performed well against the Monte Carlo gradient ascent policy.

3.1.8 Final - Monte Carlo gradient ascent policy vs. IPW gradient ascent policy

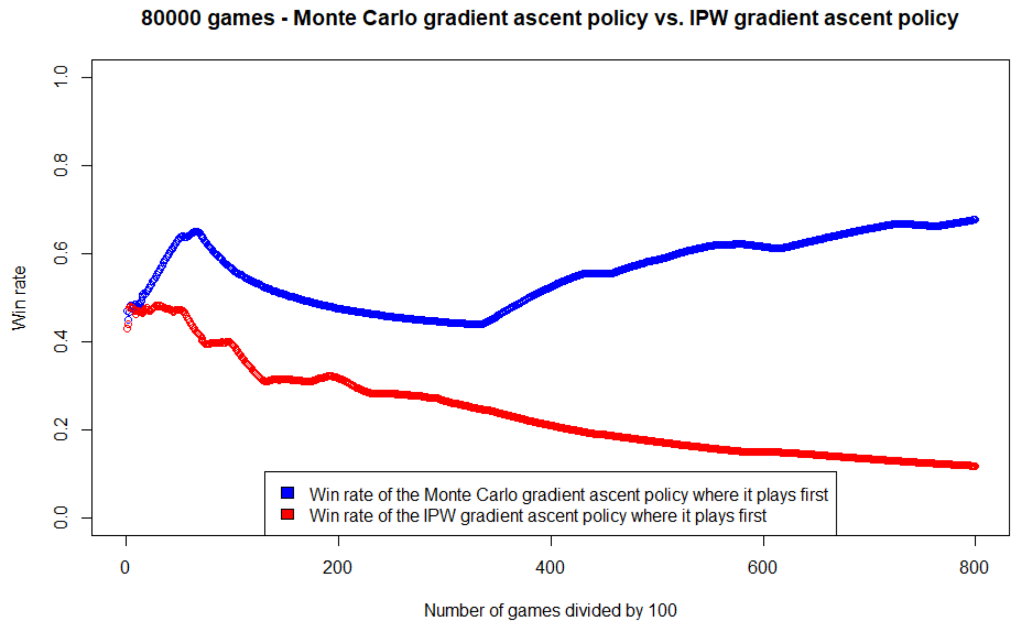


Figure 25: The win rate of the Monte Carlo gradient ascent policy with $\alpha = 0.4$ and the IPW gradient ascent policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the Monte Carlo gradient ascent policy played first in all games against the IPW gradient ascent policy. For the red line the IPW gradient ascent policy played first in all games against the Monte Carlo gradient ascent policy.

In Figure 25 we see that the Monte Carlo gradient ascent policy finds an early exploit on the IPW gradient ascent policy. However, at approximately 10,000 games the IPW gradient ascent policy catches up. We can see how the win rate of the Monte Carlo gradient ascent policy drops slowly, since the IPW gradient ascent policy is slowly improving and most likely causing draws. However, the Monte Carlo gradient ascent policy is able to learn the optimal strategy, whereas the IPW gradient ascent policy cannot. After playing about 38,000 games, the Monte Carlo gradient ascent policy slowly finds new exploit that the IPW gradient ascent policy cannot comprehend. This causes a major lead.

Match result: Monte Carlo gradient ascent policy wins the tournament!

Monte Carlo gradient ascent policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	54166	1004	24830	53162
Plays second	28649	9364	41987	19285

IPW gradient ascent policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	9364	28649	41987	-19285
Plays second	1004	54166	24830	-53162

3.1.9 Bronze match - Monte Carlo ϵ-greedy policy vs. IPW gradient ascent policy

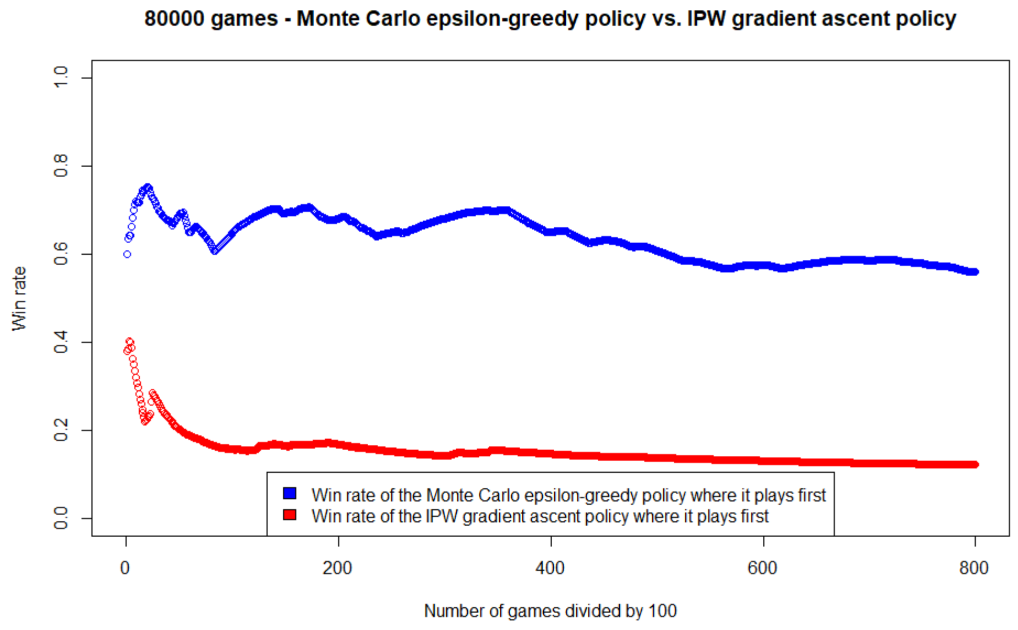


Figure 26: The win rate of the Monte Carlo ϵ-greedy policy with $\epsilon = 0.05$ and the IPW gradient ascent policy. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played for each starting position was 80,000. For the blue line the Monte Carlo ϵ-greedy policy played first in all games against the IPW gradient ascent policy. For the red line the IPW gradient ascent policy played first in all games against the Monte Carlo ϵ-greedy policy.

In Figure 26 we see that the Monte Carlo ϵ-greedy policy has the upper hand against the IPW gradient ascent policy. This is due to the fact that the Monte Carlo ϵ-greedy policy learns faster. Even though the policies somehow even out when it comes down to long-term performance this won't make a difference when only 80,000 games are played.

Match result: Monte Carlo ϵ-greedy policy wins the second place of the tournament!

Monte Carlo ϵ-greedy policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	44769	4340	30891	40429
Plays second	9555	9702	60743	-147

IPW gradient ascent policy				
	Wins	Losses	Draws	Cumulative gain
Plays first	9702	9555	60743	147
Plays second	4340	44769	30891	-40429

3.1.10 Conclusion and analysis of the tournament

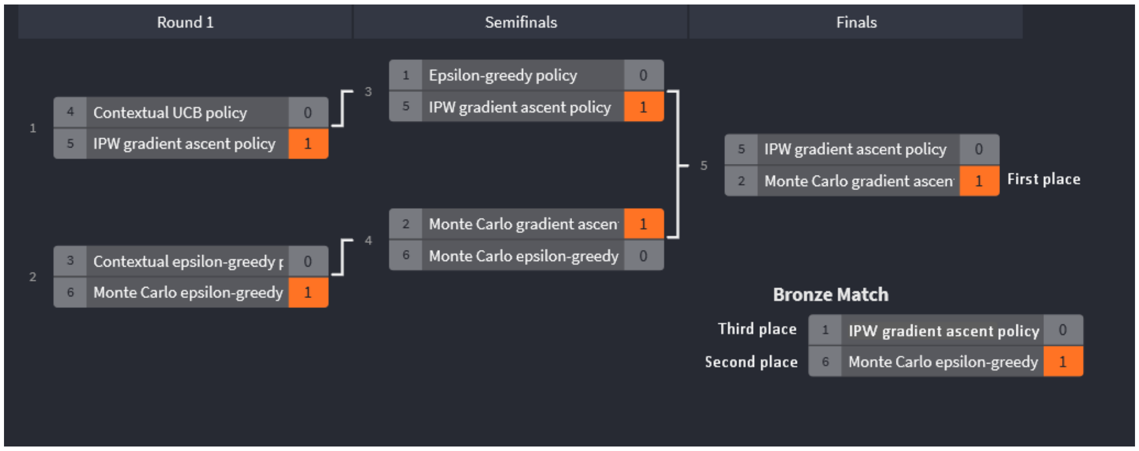

Round 1
Semifinals
Finals
1
4 Contextual UCB policy 0
5 IPW gradient ascent policy 1
2
3 Contextual epsilon-greedy p 0
6 Monte Carlo epsilon-greedy 1
3
1 Epsilon-greedy policy 0
5 IPW gradient ascent policy 1
4
2 Monte Carlo gradient ascen 1
6 Monte Carlo epsilon-greedy 0
5
5 IPW gradient ascent policy 0
2 Monte Carlo gradient ascen 1
First place
Bronze Match
Third place 1 IPW gradient ascent policy 0
Second place 6 Monte Carlo epsilon-greedy 1

Figure 27: The tournament has concluded. The Monte Carlo gradient ascent took the first place, followed by the Monte Carlo ϵ-greedy policy in the second place. IPW gradient ascent policy took the third place.

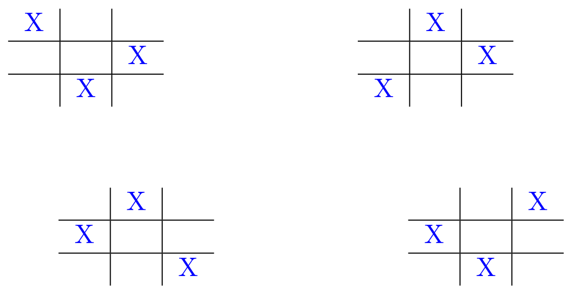
The number of games in each match seems to play an important role on the outcome of a tournament. There might be policies with an excellent long-term performance but a lower early learning speed compared to the opponent. We probably could have expected a much different outcome, if the number of games per match was 20,000. For example, the non-contextual ϵ-greedy policy would probably have come close to beat the winner of the tournament, even though a non-contextual policy is unsuited for Tic-Tac-Toe. Perhaps if one has a small training set an unsuitable algorithm might be better than an advanced one. For Tic-Tac-Toe the size of the training set does not matter much unless one has a very small memory capacity. If the memory capacity was small enough it could be impossible to store all the information needed to properly train a non-contextual algorithm. Perhaps, if the game board was a lot bigger than 3×3 this might have been a concern. However, under the conditions of this thesis the Monte Carlo gradient ascent policy is the most superior among the other policies in the tournament. The Monte Carlo gradient ascent policy is superior to the IPW gradient ascent policy since it uses an action-value function and that it utilizes all information given an iterative approach. The Monte Carlo ϵ-greedy policy is superior to the IPW gradient ascent policy since it learns faster due to the nature of an ϵ-greedy approach. The ϵ-greedy policy chooses a random action more often since the exploration is not based on a preference. Therefore, the ϵ-greedy approach finds a rewarding action faster than a gradient ascent method.

3.2 Changing winning conditions

So far, we have seen how the decision policies perform when playing Tic-Tac-Toe under normal winning conditions. To study the adaptability of the decision policies, we can make the decision policies play a series of games where the winning conditions are changed. I will carry out two experiments, one where the winning conditions are radically different from the normal winning conditions, and one where the winning conditions are more similar to the normal winning conditions.

3.2.1 Experiment with radically different winning conditions

All decision policies (except Left policy and Random policy) will play 200,000 games against the Random policy. First the decision policies plays first for 100,000 games, afterwards the decision policies play second. For every 30,000'th game the winning conditions will change. For all the games in the intervals $[30000, 60000]$ and $[90000, 100000]$ we will use the following winning conditions below.



For all other games we will use the normal winning conditions. For the new winning conditions there are four winning positions. These winning positions were chosen since they are very different from the normal winning positions, and that might make it easier to study the adaptability of the decision policies.

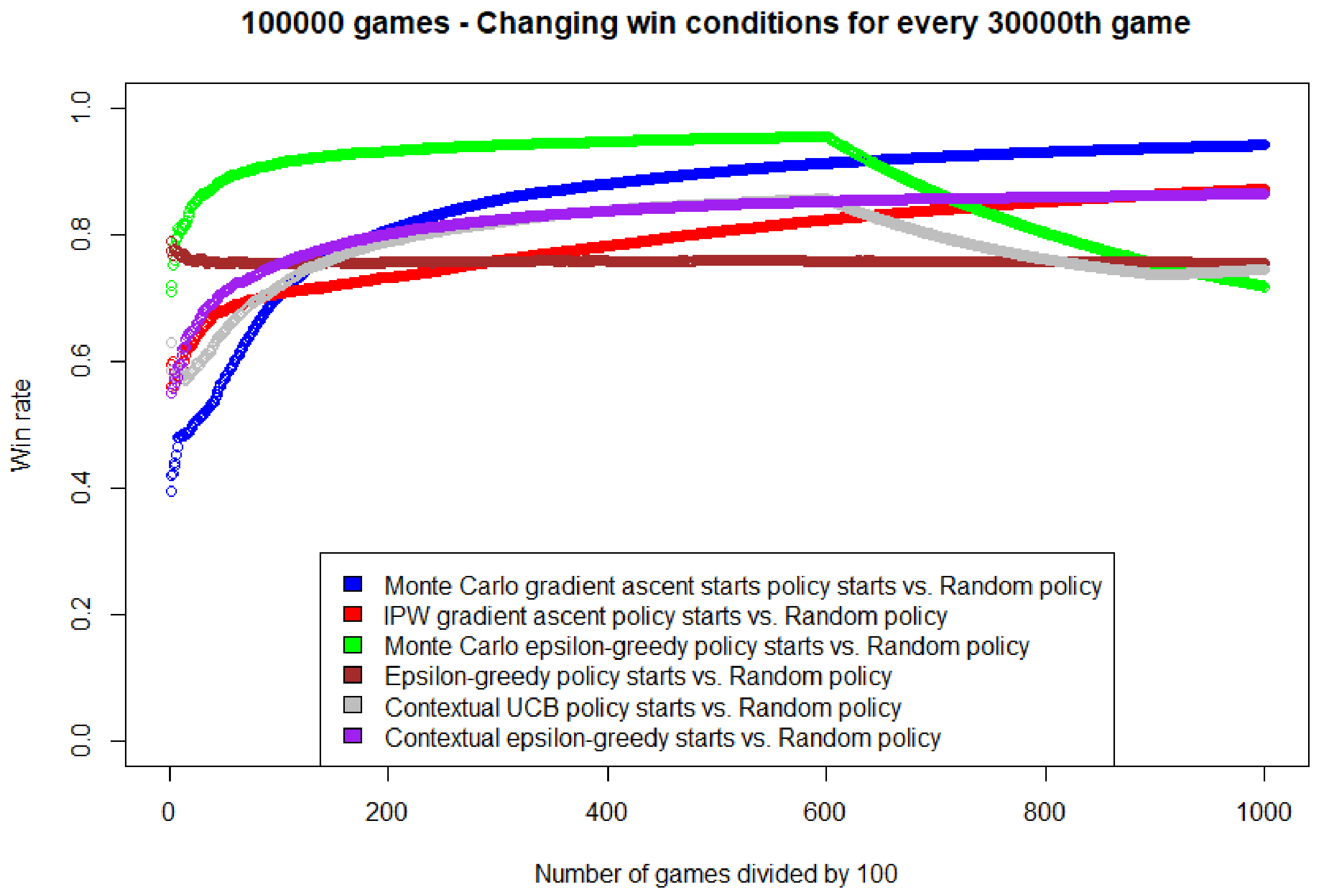


Figure 28: The win rate of the decision policies (except the IPW gradient ascent policy) when playing first under changing winning conditions that are radically different. The winning conditions switched every 30,000'th game. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 100,000.

We see that all policies in Figure 28 are unaffected by the change in winning conditions, except for the Monte Carlo ϵ-greedy policy and the contextual UCB policy. It is hard to say if this is more or less a coincidence. Even though both policies learn an effective strategy and are unaffected by the first change of winning conditions, they drop rapidly in performance after the second change.

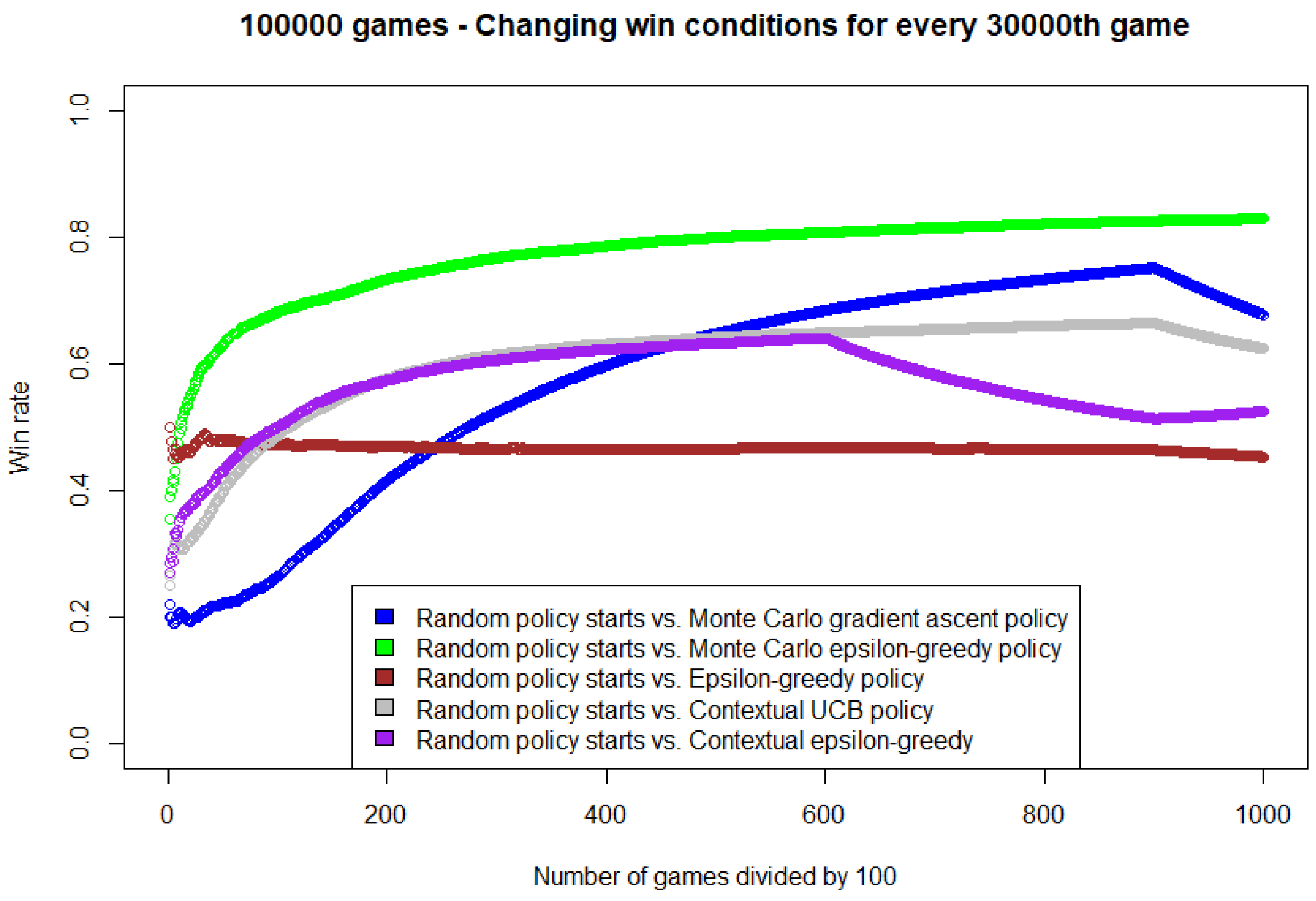


Figure 29: The win rate of the decision policies when playing second under changing winning conditions that are radically different. The winning conditions switched every 30,000'th game. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 100,000.

We see that all policies in Figure 29 are more or less unaffected by the change in winning conditions until after a certain point. Surprisingly, the Monte Carlo ϵ-greedy policy is completely unaffected, unlike what we saw in Figure 28. All the policies that were unaffected in Figure 28, are now performing worse when the winning conditions are changed. It seems to be a coincidence when it comes down to which policies are affected. Perhaps if more games were played we could see a clearer tendency. I decided to run an extra 300,000 games where all policies play first. This time, the winning conditions will only switch once when 150,000 games have been played.

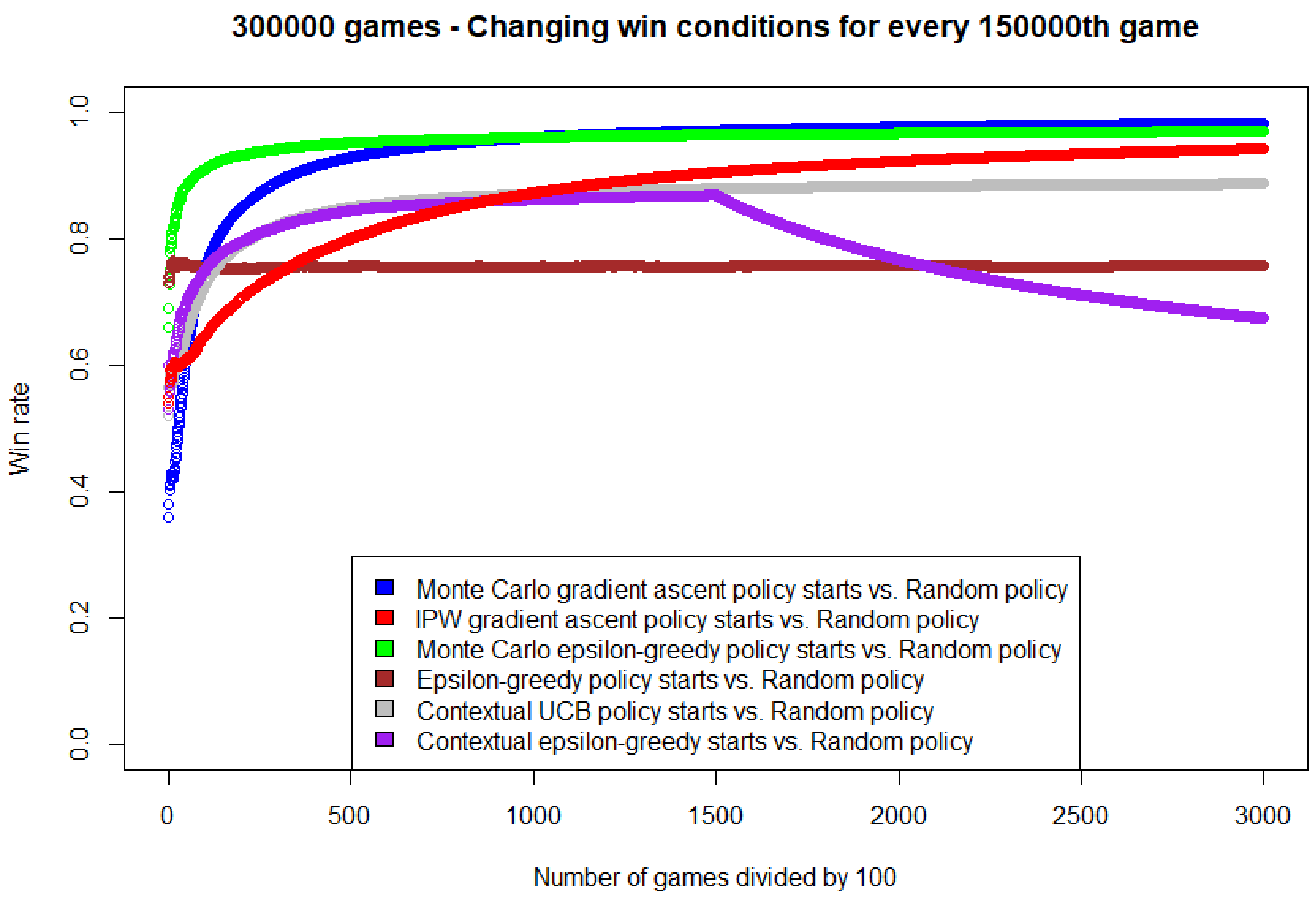
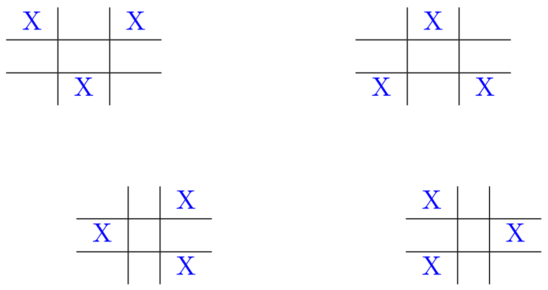


Figure 30: The win rate of the decision policies when playing second under changing winning conditions that are radically different. The winning conditions switched every 30,000'th game. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 100,000.

In Figure 30 we see that the contextual ϵ-greedy policy is the only policy affected by the changing winning conditions. It seems that it will require many games of new experience until the contextual ϵ-greedy policy adapts to the new conditions.

3.2.2 Experiment with similar but different winning conditions

All decision policies (except Left policy and Random policy) will play 200,000 games against the Random policy. First the decision policies plays first for 100,000 games, afterwards the decision policies play second. For every 30,000'th game the winning conditions will change. For all the games in the intervals $[30000, 60000]$ and $[90000, 100000]$ we will use the following winning conditions below:



For all other games we will use the normal winning conditions. For the new winning conditions there are four winning positions. These winning positions were chosen since they are different but more similar to the normal winning positions. This might reveal something interesting about the adaptability of the decision policies.

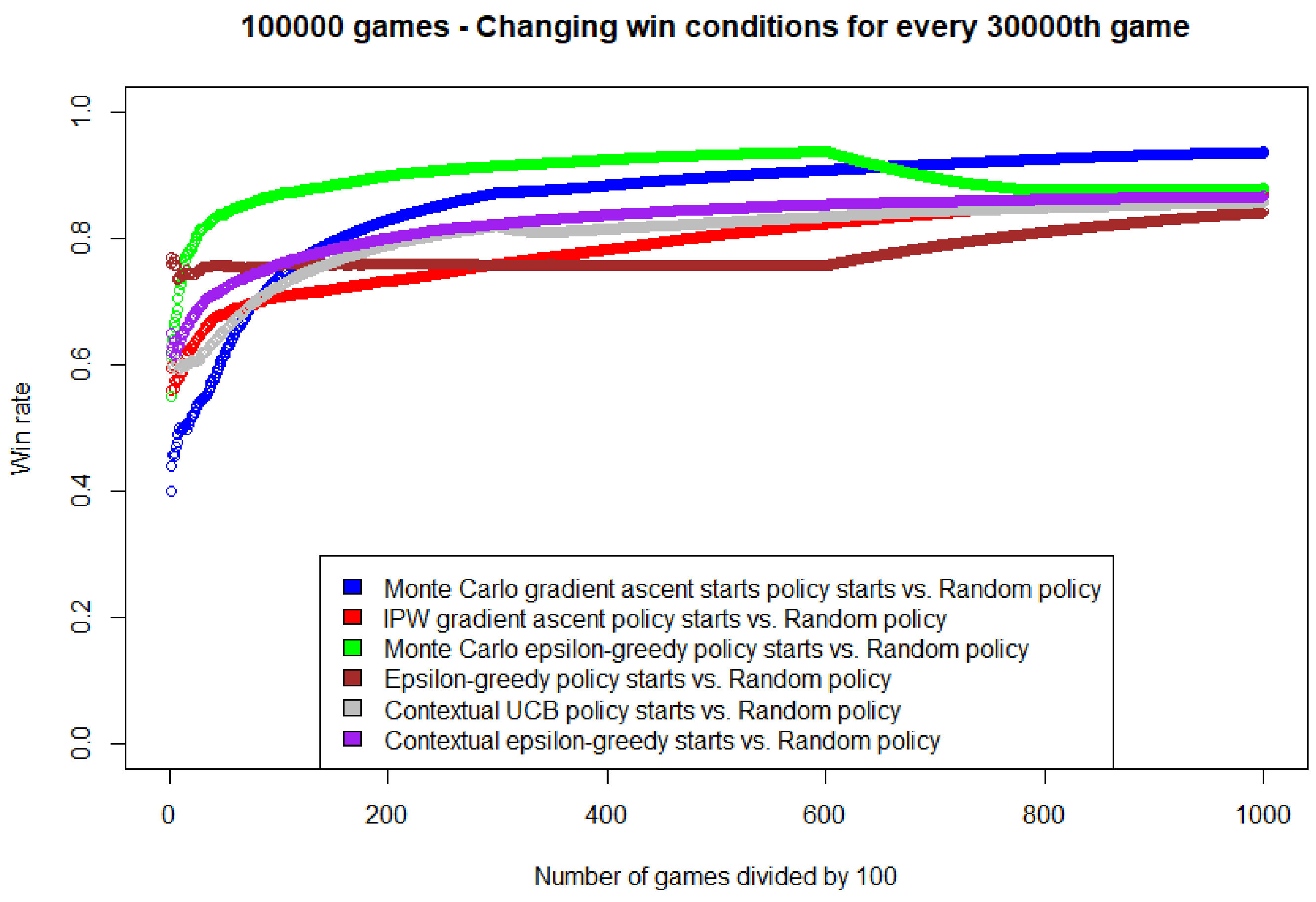


Figure 31: The win rate of the decision policies when playing first under changing winning conditions that are more similar but different. The winning conditions switched every 30,000'th game. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 100,000.

In Figure 31 we see that some policies are not even noticeably affected by the changing winning conditions. It seems that the learning rate of the Monte Carlo gradient ascent policy becomes slightly more stale after the 30,000'th game. The Monte Carlo ϵ-greedy strategy is negatively affected when the normal winning conditions are reinstated at the 60,000'th game. However, all policies seems to be adapting well. The ϵ-greedy policy even seems to play better when the winning conditions are switched back to normal in the 60,000'th game.

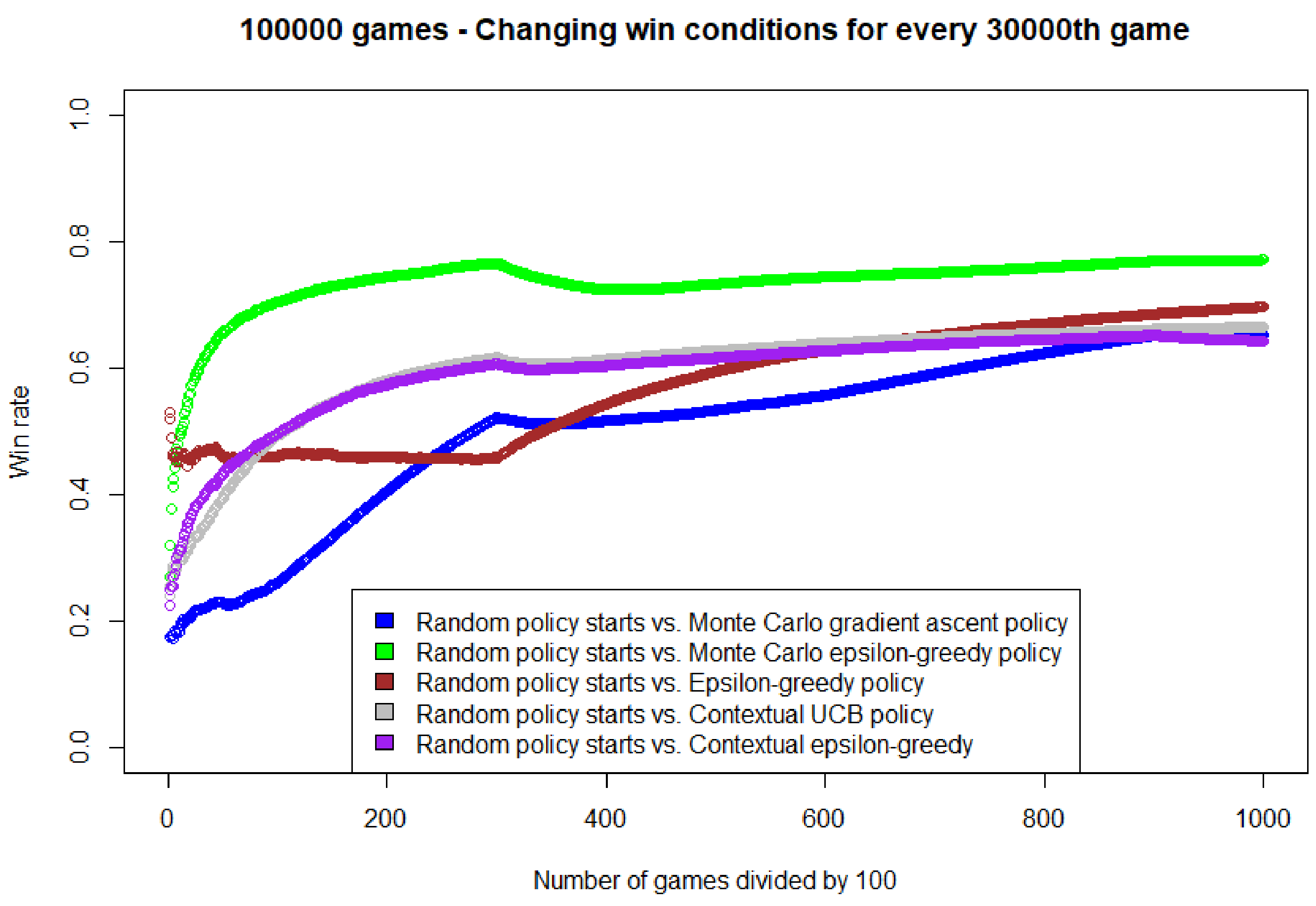


Figure 32: The win rate of the decision policies (excluding the IPW gradient ascent policy) when playing second under changing winning conditions that are more similar but different. The winning conditions switched every 30,000'th game. For every 100'th game we calculate the win rate as the percentage of wins out of the most recent 100 games. On the first axis we have the number of games divided by 100, and on the second axis we have the win rate. The total number of games played was 100,000.

In Figure 32 we see the win rate of the decision policies when they play the second move. It is more clear that the winning conditions change at the 30,000'th game. It does not seem like any policy is much worse at adapting than any other policies. If we look at the ϵ-greedy policy it actually seems to improve drastically after the first change of winning conditions. The improvement does not go away even though the winning conditions are changed back.

3.2.3 Conclusion of the changing winning conditions experiment

It seems to be coincidental whether or not a policy is affected by the changing winning conditions when the new conditions are radically different from the original. This also seems to be the case for winning conditions that have similarities to the original conditions. In some cases policies would drop in win rate quite rapidly and then slowly begin to flatten out. It is hard to explain why this behaviour happens since it is very coincidental. Sometimes the change does not make any difference, and sometimes the change causes a relatively huge backlash that is hard to recover from. An explanation for the drop in win rate could be that policies continue to play the same actions as before the change.

However, this time all of these actions lead to draws. Thus, it takes longer to find the new optimal strategy. This could also explain why the win rate for some policies goes straight back up once the old winning conditions are reinstated.

4 Conclusion

Several different types of reinforcement learning techniques have been implemented and compared, these are referred to as decision policies. A non-contextual decision policy have been implemented, even though the Tic-Tac-Toe problem is a contextual problem. A non-contextual technique, when applied to a contextual problem, can be effective on for the short term. However, unless the training data or memory capacity is very small, a non-contextual decision policy should be avoided for contextual problems such as Tic-Tac-Toe.

Contextual decision policies have been implemented and compared, these generally performed better than the non-contextual decision policy. For Tic-Tac-Toe, a decision policy must be contextual if the policy should be able to approximate an optimal strategy. However, it is not enough to simply implement a contextual decision policy. If no reward is received after performing an action, there will be no learning. A Tic-Tac-Toe game cannot be decided in the first few turns, therefore a contextual decision policy must also consider how to achieve a well-defined reward for the actions in the first few turns.

Two decision policies have been implemented that are both contextual and have a well-defined reward for each action. The policies are both based on the Monte Carlo approach. However, the decision policies use different techniques for deciding between exploration and exploitation. Only one of these two policies managed to approximate an optimal strategy. The technique that managed to approximate an optimal strategy, was the stochastic gradient ascent technique. Surprisingly, it even managed to achieve a better cumulative gain by losing some games rather than playing the optimal strategy where one never loses. The other Monte Carlo policy used an ϵ-greedy approach to decide between exploration and exploitation. Based on the results in this thesis, we can conclude that an ϵ-greedy approach has a faster early learning, than a stochastic gradient ascent approach. However, a stochastic gradient ascent approach will reach a higher long-term cumulative gain. When using the ϵ-greedy approach, the learning curve will flatten earlier than when using a stochastic gradient ascent approach.

Jonas Peters' version of stochastic gradient ascent using inverse probability weighting (IPW) in [3, p. 10] have been compared to the other policies in this thesis. The IPW stochastic gradient descent policy is also contextual and has a well-defined reward for each action. In terms of learning speed and long-term cumulative gain, the Monte Carlo stochastic gradient ascent policy is superior to the IPW stochastic gradient ascent policy.

Based on the results in this thesis, a gradient ascent approach is expected to perform better than an ϵ-greedy approach for the long-term cumulative gain. However, it cannot be concluded whether or not this is true for the IPW gradient ascent policy as well.

The policies have been tested under a changing environment. The policies train under normal winning conditions, then after some interval of games the winning conditions change. It seems to be coincidental if a policy is affected by the change. In one case the policy might drop rapidly in win rate, and in another case you cannot tell that the winning conditions have changed.

The computational complexity has been analyzed using asymptotic notation. The running time is $O(|A| \cdot R)$ for all policies, except for the IPW stochastic gradient ascent policy. The inverse probability weighted gradient ascent policy is implemented with a running time of $O(|E|^2 \cdot R)$, however, in practice it runs relatively fast since data is erased and a policy update happens once every 500'th episode.

5 For future research

The Monte Carlo stochastic gradient ascent policy is implemented as an on-policy method. It would be interesting to implement an off-policy version using an action-value function instead of a cumulative reward. This could be done by modifying the inverse probability weighted stochastic gradient ascent policy, since this policy is already an off-policy method.

There are many more known decision policies, it would be interesting to implement more to see how they perform. It would also be interesting to run more games when training the Monte Carlo ϵ-greedy policy and the inverse probability weighted stochastic gradient descent policy, to see if they can approximate an optimal strategy.

References

[1] Bottou, J. Peters, J. Quiñonero-Candela, D. X. Charles, D. M. Chickering, E. Portugaly, D. Ray, P. Simard, and E. Snelson. Counterfactual reasoning and learning systems. *Journal of Machine Learning Research 14:3207-3260*, 2019. http://jmlr.org/papers/volume14/bottou13a/bottou13a.pdf.

[2] J. L. Lihong Li, Wei Chu and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. *In Proc. of the 19th international conference on World wide web Pages 661-670*, 2019. http://rob.schapire.net/papers/www10.pdf [Accessed: March 22nd, 2019].

[3] J. M. Peters. Learning to play tic-tac-toe. *Made available through personal communication*, 2019. [Accessed: May 7th, 2019].

[4] A. Rønn-Nielsen and E. Hansen. Conditioning and markov properties, 2018. http://web.math.ku.dk/noter/filer/beting.pdf [Accessed: June 14th, 2019].

[5] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction (complete draft), 2019. http://incompleteideas.net/book/bookdraft2017nov5.pdf [Accessed: April 22nd, 2019].