MENACE and Q-Learning Playing Noughts and Crosses

Bram Pulles

Department of Science, Radboud University, Nijmegen, NL
Email: borroot@live.nl

This paper describes two trial-and-error algorithms which learn to play the game of Noughts and Crosses. These two methods are called MENACE and Q-learning. MENACE was initially constructed physically from matchboxes and coloured beads and was subsequently simulated in essence by a program for a Pegasus 2 computer. This article will describe a modern implementation of the original device for MENACE. This article will also describe two other algorithms which can be used to play Noughts and Crosses perfectly, namely minimax and negamax. These two algorithms are used to teach the trial-and-error algorithms. The parameters governing the adaptive behaviour of the trial-and-error methods are described. Furthermore preliminary observations on the performance of the methods are briefly reported.

Keywords: Data Mining; Machine Learning; Reinforcement Learning; MENACE; Q-learning; Minimax; Negamax

Received December 5, 2019; revised December 5, 2019

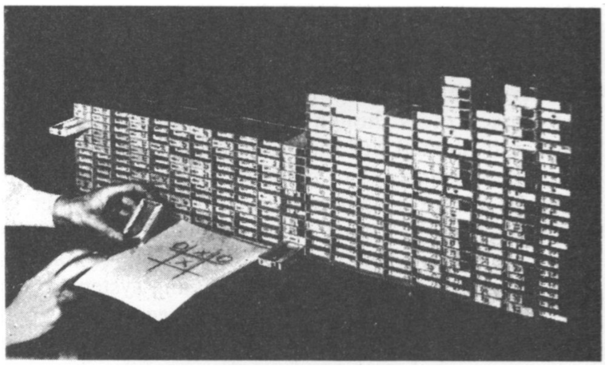

FIGURE 1. The original matchbox machine - MENACE

1. INTRODUCTION

This article describes two reinforcement learning algorithms, MENACE and Q-learning, and two perfect play algorithms, minimax and negamax, for Noughts and Crosses. In this article these algorithms will be described mathematically, their complexities will be analysed and their performance will be reported.

2. BACKGROUND

MENACE stands for Matchbox Educable Noughts And Crosses Engine and was first developed in 1961.[1] In figure 1 you can see the original physical version made by Donal Michie. He played against this device for a whole weekend to get the data for his research. A computer version of MENACE was developed for the Pegasus 2 shortly after the physical version was created. The computer version could play about one game per second on the Pegasus 2, as a comparison my implementation simulates tens of thousands of games per second, but it is still way better than playing against the device yourself.

Q-learning is a more modern reinforcement algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances.[2] Q-learning and MENACE are very alike, more about this in section 3.1.

All of the code for this project can be found at `github.com/borroot/menace` and a very extensive Javadoc documentation, for all the functions, classes and nearly all variables, can be found at `borroot.github.io/menace`.

3. THE ALGORITHMS

In this section I will explain how the algorithms work. I will also talk about the complexities of the algorithms and some specifics about my implementation.

3.1. Explanations

We will start off with a description of the minimax algorithm followed by the negamax algorithm. Then we will describe MENACE and Q-learning.

3.1.1. Minimax

Minimax (sometimes MinMax, MM or saddle point) is a decision rule used in artificial intelligence, decision theory, game theory, statistics and philosophy for minimising the possible loss in a worst case (maximum loss) scenario.[3] Normally minimax is depth limited, but since Noughts and Crosses only has a maximum depth of nine I omitted the depth restriction. This makes the algorithm able to always determine the best move. This is done by calculating the minimax value for all the moves possible from the current board state and subsequently selecting the best option out of this.

The pseudo code for calculating the minimax value in a given board state is given in listing 1.

```
function minimax(node, maximizingPlayer)
  if node is a terminal node then
    return the heuristic value of node
  if maximizingPlayer then
    value := -INF
    for each child of node do
      value := max(value, minimax(child, FALSE))
    return value
  else (* minimizing player *)
    value := +INF
    for each child of node do
      value := min(value, minimax(child, TRUE))
    return value
```

Listing 1. Minimax value function.

The heuristic values here for a tie, win and loss are 0, 1 and −1 respectively. In order to make the minimax value useful in choosing the best move in a given board state we use another function. This function is given in listing 2.

```
function best_move(node)
  max := -INF
  best_moves := []
  for each child of node do
    value := minimax(child, -INF, +INF, FALSE);
    if value > max then
      max := value
      best_moves := [move from node to child]
    else if value == max then
      best_moves := best_moves + move from node to child
  return random element from best_moves
```

Listing 2. Best move function with minimax.

This function will calculate the best moves using the minimax value function and then select one of the best moves randomly.

Using this method we have to evaluate about 500.000 nodes in order to get the best option for the first move. In order to reduce the number of nodes to be evaluated we use a method called alpha-beta pruning. This way the function for calculating the minimax value stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves do not need to be evaluated further. When this is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[4] The pseudo code for calculating the minimax value using alpha-beta pruning can be seen in listing 3.

Note that the code for selecting the best move in listing 2 does not have to change. Using the new algorithm for calculating the minimax value with alpha-beta pruning we only have to evaluate about 30.000 nodes to get the best option for the first move.

```
function minimax(node, $\alpha$, $\beta$, maximizingPlayer)
  if node is a terminal node then
    return the heuristic value of node
  if maximizingPlayer then
    value := -INF
    for each child of node do
      value := max(value, minimax(child, $\alpha$, $\beta$, FALSE))
      $\alpha$ := max($\alpha$, value)
      if $\alpha$ >= $\beta$ then
        break (* $\beta$ cut-off *)
    return value
  else (* minimizing player *)
    value := +INF
    for each child of node do
      value := min(value, minimax(child, $\alpha$, $\beta$, TRUE))
      $\beta$ := min($\beta$, value)
      if $\alpha$ >= $\beta$ then
        break (* $\alpha$ cut-off *)
    return value
```

Listing 3. Minimax value function with alpha-beta pruning.

This results in a great increase in speed and will be very helpful when the reinforcement algorithms play thousands of games against minimax.

3.1.2. Negamax

Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game. This algorithm uses the fact that

$$\max(a, b) = -\min(-a, -b)$$

to simplify the implementation of the minimax algorithm. More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on turn looks for a move that maximises the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether it is A or B's turn. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.[5]

The pseudo code for calculating the negamax value, which is equal to the minimax value, in a given board state is given in listing 4.

```
function negamax(node, color)
  if node is a terminal node then
    return color * the heuristic value of node
  value := -INF
  for each child of node do
    value := max(value, -negamax(child, -color))
  return value
```

Listing 4. Negamax value function.

The heuristic function is exactly the same as before so 0, 1 and −1 for a tie, win and loss respectively. The color represents the players turn and has either a value of 1 for the maximising player or −1 for the minimising player.

Again we have a separate function for calculating the best move in a given board state. This function is

```
function best_move(node)
  max := -INF
  best_moves := []
  for each child of node do
    value := -negamax(child, -INF, +INF, -1)
    if value > max then
      max := value
      best_moves := [move from node to child]
    else if value == max then
      best_moves := best_moves + move from node to child
  return random element from best_moves
```

Listing 5. Best move function with negamax.

given in listing 5. As you can see the only difference between listing 5 and listing 2 is the function which is called when calculating the evaluation of the board.

Now we will improve the algorithm for calculating the negamax value in exactly the same way as we improved the calculation of the minimax value so by implementing an alpha-beta pruning version. The new improved version can be seen in listing 6.

```
function negamax(node, α, β, color)
  if node is a terminal node then
    return color * the heuristic value of node
  value := -INF
  for each child of node do
    value := max(value, -negamax(child, -β, -α, -color))
    α := max(α, value)
    if α >= β then
      break (* cut-off *)
  return value
```

Listing 6. Negamax value function with alpha-beta pruning.

Now it is clear why we want to use negamax instead of minimax. As you can see listing 6 is significantly shorter than listing 3. Although one could argue that minimax is easier to understand than negamax.

3.1.3. MENACE

MENACE is a simple reinforcement algorithm. For every 765 *essentially distinct* board states we have a separate imaginary box with beads inside. The words "essentially distinct" are emphasised, because we take all the eight symmetries of the Noughts and Crosses board into account[1] and identify them as one and the same board state. Each box contains an assortment of differently identified beads. The different beads correspond to unoccupied squares where a move could be made. The identification of the beads is seen in figure 2. The beads are identified by numbers.

0	1	2
3	4	5
6	7	8

FIGURE 2. Identification of the squares by numbers.

[1]The eight symmetries are: the identity, flip over vertical axis, flip over horizontal axis, flip over main diagonal, flip over other diagonal, rotate 90 degrees, rotate 180 degrees and rotate 270 degrees.

Now imagine we want to play against this machine. In order to determine the current move to be made we take the corresponding box and select a random bead from this box. The bead which is chosen is also the move played by MENACE. We now put the box aside and remember what move we chose. This process will be repeated until the game terminates. At this point the learning will start. We tell MENACE if it won, tied or lost. All the results have a certain reward. A loss will always result in the removal of the chosen beads from the boxes. A win or tie will always result in an addition of a specific amount of beads to the boxes. This way the next round MENACE is more/less likely to choose moves according to the feedback it got previously. This is the core of the algorithm and enables it to learn by trial-and-error.

3.1.4. Q-learning

Q-learning works just like MENACE in the sense that for every board state Q-learning assigned a value to all the possible actions from this state. However in Q-learning these values are not restricted to integers and Q-learning will always choose the best option (the highest value) instead of choosing one at random.

When Q-learning is initialised it will create a big table with all the possible board states representing the different rows. The columns will be represented by all the possible actions. An explicit example can be seen in table 1.

		Actions								
		0				...				8
	1	0	0	0	0	0	0	0	0	0
	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
States	300	0	0	0	0	0	0	0	0	0
	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.	.	.
	765	0	0	0	0	0	0	0	0	0

TABLE 1. Q-learning table when just initialised.

When this table is created we can already start playing Noughts and Crosses. For every move Q-learning will select one of the best moves. If there are multiple moves with the same maximum value then randomly choose one of these, and play this one. When a move has been made the board state is saved together with the action which was taken by the algorithm. This will continue until the board reaches a terminating state, then the learning will start.

The formula for learning looks as follows

$$Q^{new}(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max\ Q(s_{t+1}, a))$$

This might look quite daunting at first, but I will break it down piece by piece.

Q is a function which takes in a state and an action and returns the corresponding value from the table shown before. So $Q(s_t, a_t)$ is the value at time t in the table for state s and action a. The type of Q is thus $Q : S \times A \rightarrow \mathbb{R}$.

α represents the learning rate of the Q-learning algorithm. The learning rate determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

r_t represents the reward which followed from making the actions which were made. A reward of 1 represents a win, 0 a tie and -1 a loss for the Q-learning algorithm.

γ represents the discount rate and determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards while a factor approaching 1 will make it strive for a long-term high reward.

$max\ Q(s_{t+1}, a)$ represents the estimate of the optimal future reward. This is calculated by taking the maximum value from all the possible actions a in state s_{t+1}. Where s_{t+1} the new state is when the move a_t has been made in state s_t.

As a last remark, for all the final states s_f, $Q(s_f, a)$ is never updated, but is set to the reward value r observed for the state s_f.

So when the game reaches a terminating state all the cells in the table of Q will be updated according to the formula. This will make the Q-learning algorithm able to learn. After playing for some time the table will look for example as seen in table 2.

		Actions				
		0		. . .		8
States	1	-12.04	-9.67	. . .	-6.23	11.21
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	300	-9.10	-8.20	. . .	-7.33	-6.08
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	765	0.91	14.96	. . .	-6.28	-9.54

TABLE 2. Q-learning table when it played some games.

We still miss one last thing in the Q-learning algorithm. Consider the situation were the algorithm will win during the first game due to a mistake of the opponent. Then even if the moves made by the Q-learning algorithm were quite bad they will still get a positive reward. Now the next turn the algorithm will make the exact same, possibly stupid, moves since this one was rewarded previously and has the highest value. This makes it possible that the Q-learning algorithm would settle for a local maximum instead of the general maximum. So the algorithm could for example settle for playing a tie every game even though it might be able to win sometimes as well. In order to prevent this from happening we introduce another variable ϵ.

The ϵ value stands for exploration. This value will range from 0 to 1 where 0 means the algorithm will never explore other nodes than the maximum node and 1 means it will effectively ignore the maximum node and always make a random move. So every time before a move is chosen from the table we generate a random number between 0 and 1 and if this random number is smaller than ϵ then we make a random move instead of taking the best currently known move.

To even further improve the Q-learning algorithm we also add a degradation of the ϵ value. At the start of the learning process we want to explore a lot of nodes. Once we are learning for quite some time we are pretty sure that we have the right solution so exploration is not desired anymore, since we want to converge to the right solution. So we specify a start and end value for exploration. The start value is an integer which represents the n'th move were we want to start degradation of the exploration value. The end value specifies the n'th round were we want to have an exploration of 0. In between the start and end value the exploration will decrease linearly every round.

3.2. Complexity

The time complexity for both MENACE and Q-learning depends on the maximum amount of actions which can be taken from a state.[2] It also depends on the maximum amount of moves which can be made during one game by one of the players.[3]

The maximum amount of actions is relevant when MENACE and Q-learning need to make a move. If we look at both algorithms we can see that the time complexity of the *move* functions are equal to $O(n)$, with n being the maximum amount of moves per state. The time complexities for the *learn* functions are a little bit different for both of the algorithms. For MENACE this complexity is $O(m)$ where m is the maximum amount of moves which can be made by any player during the game. For Q-learning this complexity is $O(n \cdot m)$. This gives use a total time complexity of $O(n + m)$ for MENACE and $O(n + n \cdot m)$ for Q-learning. Note that both n and m will never really get big, above about 1000, thus making the time complexity quite good for both algorithms. Note furthermore that the the lookup in the table for the current state is in constant time because of the usage of a hash-map in both algorithms.

[2]To give some examples, this amount equals 9 for Noughts and Crosses, 7 for Connect Four and 361 for Go.

[3]Some more examples, this amount is 5 for Noughts and Crosses and 21 for Connect Four.

However the space complexity of the algorithms is not so good. Both MENACE and Q-learning effectively put a table in storage proportional to the amount of states times the amount of actions. For Nought and Crosses this is no problem since this would only result in $765 \cdot 9 = 6885$ cells which just contain either integers or floats. If the game would have a bigger state space complexity then this could become a problem. Take for example the game Connect Four which already has a state space complexity of about 10^{13} and a maximum amount of 7 possible moves from every state. If every cell would have an integer of 8 bytes then the total amount of storage needed will be $8*7*10^{13}$ bytes which equals to about 560 TB. So these two algorithms are only useful for games with a low state space complexity.

3.3. Implementation

Now we will dive into some specifics of my implementation.

There are two situations which can occur inside of the MENACE machine of particular interest. The first one is MENACE cannot make a move. This happens when the box corresponding to the current state is completely empty because MENACE was punished a lot. In my implementation I made an exception for this such that MENACE immediately resigns from the current game if this situation occurs.

The second situation is when MENACE cannot make a move at the start. So if MENACE is player one then this means it cannot make a move on the empty board because the corresponding box is empty. If MENACE is player two then this means it cannot make a move on any of the boards, there are three distinct ones, corresponding to the starting move for player two. In this case we declare MENACE dead and we stop the whole training. Since if we would not then MENACE would just immediately resign for all the left over games and this would give confusing results.

4. RESULTS

Now I will talk about the results of training the reinforcement algorithms. Both algorithms are trained against a random player and a perfect player. Minimax and negamax play equivalently and can thus both be used to represent the perfect player.

The performance of the algorithms will mainly be showed by graphs which represent the amount of wins/ties/losses over time of the algorithm. For every win the graph will go one up, for every loss it will go one down and for every tie it will stay on the same level. Some of the graphs contain multiple training sessions which are represented by grey lines. The black lines in these graphs is the average performance over all the training sessions.

Note that all the training will be assumed to not alternate turns. This is because the alternations of turns is the same as training two distinct players and will thus only slow down the learning process.

We will also assume that all the figures are relative to either MENACE or Q-learning and *not* the algorithm they are trained against.

Lastly I want to note that all of the graphs are automatically generated using the main Java program which is connected through Bash with a Python program to show the plots.[4]

4.1. MENACE

We will use a convention here to abbreviate the variables of MENACE. The setup will be represented as $M(i, r_w, r_t)$ with i being the initial amount of beats per move per box, r_w is the reward for a win and r_t is the reward for a tie. The punishment for a loss will always be 1 since we are always sure when we made a move that there is 1 bead in the box. However can not be sure that there are more than one beads in a box which would give complications with the removal of more beads.

First we train MENACE, $M(1, 2, 1)$, against the random player see figure 3. Here you can see that MENACE is struggling a little at the start, but quickly finds some good methods and starts winning or playing ties. To play against a random player it is advantageous to choose a low initial amount of beads since the algorithm will most likely not get punished often by a random player and thus has a very small chance to die. The low amount of beads makes the learning go faster. To illustrate this see figure 4 were we have $M(10, 2, 1)$ against the random player. We chose $i = 10$ here to see the effect strong and easily recognisable. As you can see the player is less punished for when it loses enabling the random player to win quite a few games.

Next up we will train MENACE against a perfect player. Here we observe two things. First, we notice that r_w does not matter since MENACE will never win against a perfect player. Second, we realise that MENACE will get punished a lot at the start and thus needs to have a big value for i to prevent it from dying early on.

To illustrate the death of MENACE see figure 5. However we should also realise here that the death of MENACE very much depends on luck, ironically enough. Because if the correct moves are made at the start then this will be reinforced and MENACE might survive as you can see in figure 6. So if we do not want MENACE to die then we should chose a big value for i however you can take the risk of surviving with choosing a small value for i.

[4]The code for this can be found at `https://gist.github.com/Borroot/1a6b1257b3d3ee573894a03cdc127796`.

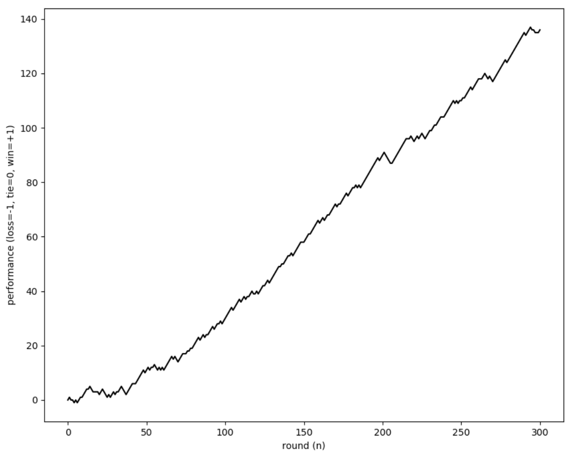

FIGURE 3. $M(1,2,1)$ against a random player. In the end MENACE won 197 games out of 300, the random player won 61 games and there were 42 ties.

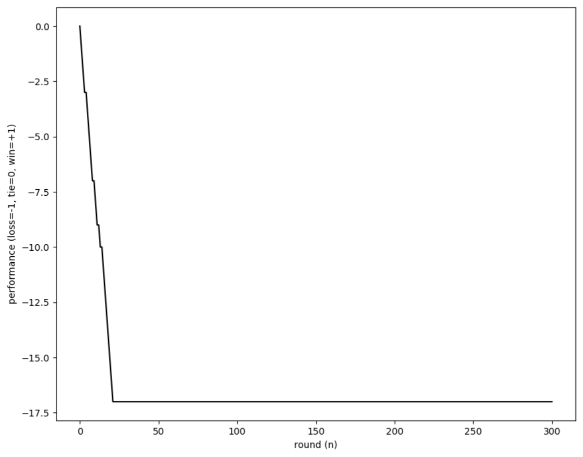

FIGURE 5. $M(1,x,2)$ against negamax. In the end MENACE won 0 games out of 300 (of course), negamax won 17 games and there were 4 ties. This means that MENACE died after playing 21 games.

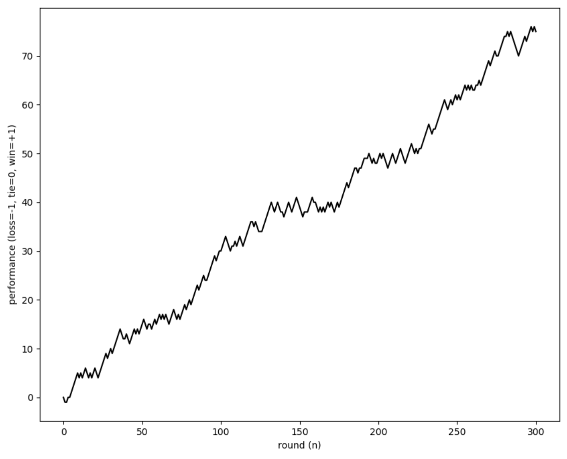

FIGURE 4. $M(10,2,1)$ against a random player. In the end MENACE won 175 games out of 300, the random player won 100 games and there were 25 ties.

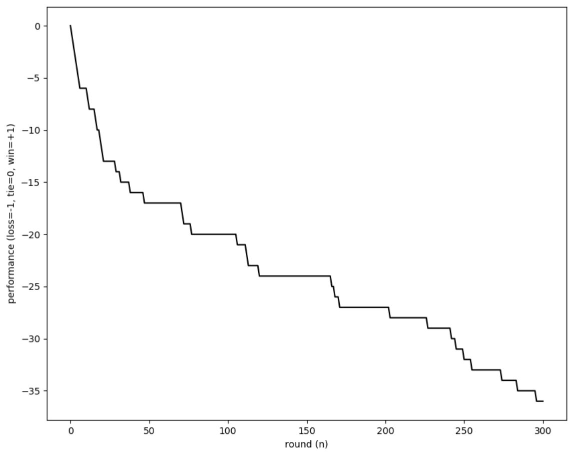

FIGURE 6. $M(1,x,2)$ against negamax. In the end MENACE won 0 games out of 300 (of course), negamax won 36 games and there were 264 ties.

At last we will let MENACE play against another MENACE. At the start both players will basically play random, but they will quickly develop a strategy and stick to it. In figure 7 you can see $M_1(1,2,1)$ play against $M_2(1,2,1)$. Something interesting happens here, M_2 does not seem to be learning since M_1 is constantly winning. This has to do with my detection of death. So what happens is M_1 will at the start constantly play, for example, in the top-left corner. M_1 is winning with this strategy due to pure luck at the start. Now at some point the box for this board will be empty at M_2 which means M_2 will always resign when M_1 plays in the top-left corner effectively making it a win for M_1. So M_1 is now adapted to M_2 to play on its weakness which is very interesting to see!

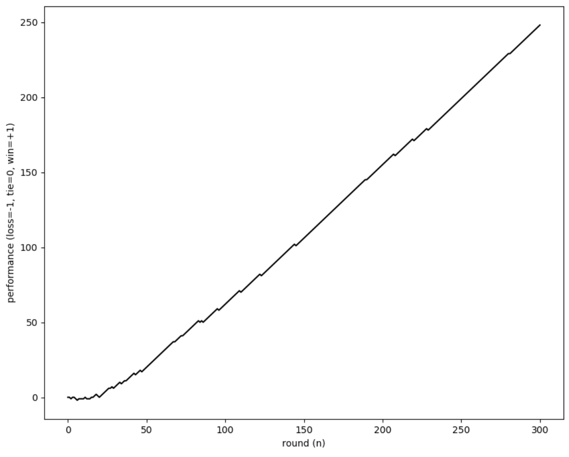

FIGURE 7. $M_1(1,2,1)$ against $M_2(1,2,1)$. In the end M_1 won 267 games out of 300, M_2 won 19 games and there were 14 ties. An illustration of the adaptive capabilities of MENACE.

4.2. Q-learning

Here we will also use a convention to abbreviate the variable which represent the Q-learning algorithm. The algorithm will be represented as follows $Q(\alpha,\gamma,\epsilon,\epsilon_s,\epsilon_e,r_w,r_t,r_l)$. The first three variables are described extensively in section 3.1.4. Furthermore ϵ_s is the start value of the exploration degradation, similarly ϵ_e is the end of the exploration degradation. Then r_t and r_w are the same as before, the reward for a tie and win respectively. Now we also have r_l which stands for *reward loss* or punishment, this is a negative value.

Before we start the training there is an important observation we make. The value for *max* $Q(s_{t+1},a)$ is calculated by taking the maximum value from all the possible actions a in state s_{t+1}. However when we are not alternating turns then all the rows corresponding to s_{t+1} will remain empty and initialised to 0. This means that the discount factor γ is effectively useless when the turns are not alternated.

We will first start with training the Q-learning algorithm against a random player. In figure 8 a common learning process is shown. As can be seen from this figure the algorithm is struggling a bit at the start, but soon finds a good play and continues to improve quite fast from here on.

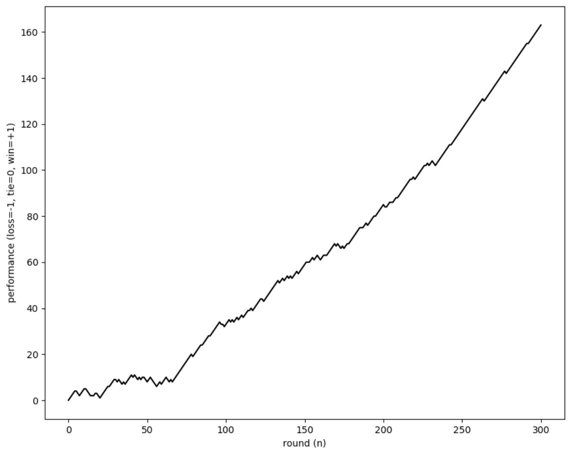

FIGURE 8. $Q(0.9,x,0.3,100,200,2,1,-1)$ against the random player. In the end Q won 217 games, the random player won 54 games and there were 29 ties.

In figure 8 we have chosen a Q with a quite high learning rate of 0.9. Now to show the effect of the learning rate we choose $\alpha = 0.1$, see figure 9. Now we can see that a low learning rate give a more stable and continuous function, especially at the start, because the algorithm does not change its behaviour as fast as before anymore.

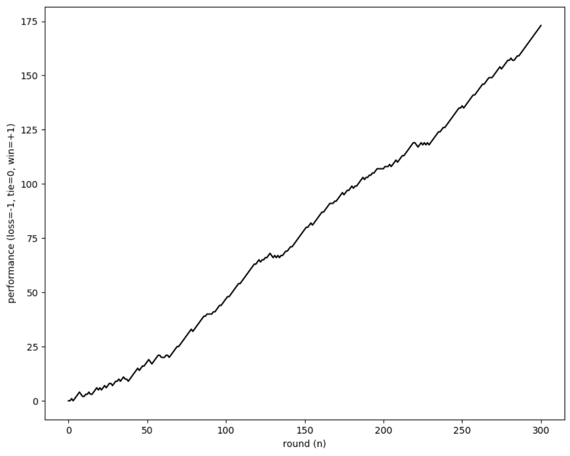

FIGURE 9. $Q(0.1,x,0.3,100,200,2,1,-1)$ against the random player. In the end Q won 209 games, the random player won 36 games and there were 55 ties.

Next up we will train Q-learning against a perfect player. Note that again the value of r_w is irrelevant, since Q-learning will never win against the perfect player.

Using a very similar setup as before but now with a lower value for exploration, start of degradation and end of degradation we get an earlier divergence. This is good since the Q-learning algorithm only has to play a tie against the perfect player. An example of this is shown in figure 10. We can see that the Q-learning algorithm performs really good against the perfect player and finds the optimal play quite fast.

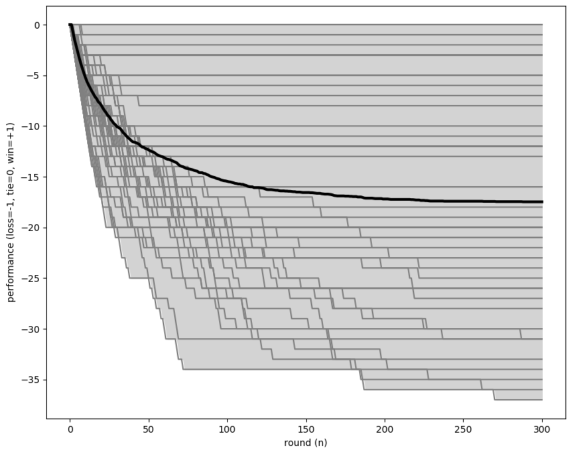

FIGURE 10. $Q(0.1, x, 0.1, 0, 50, y, 2, -1)$ against negamax. In total 50 training sessions.

Now we will show what the influence of the discount factor, γ, is. In order to do this we need to alternate turns. We will use a random player to train against since there we can also win which makes it more interesting. In figure 11 you can see Q-learning with a low discount value while in figure 12 you can see Q-learning with a high discount value. As can be seen in the figures the one with a high discount value seems to have a more steady progress and consistent behaviour, it also performs slightly better.

At last we will let Q-learning train against another Q-learning algorithm. Here they will at the start both effectively play against a random player, but they will soon converge into perfect play and constantly play a tie. In figure 13 a pretty basic setup is shown were this effect is seen. Note that the player making the first move has a slight advantage over the other player.

4.3. MENACE VS Q-learning

To conclude we will let MENACE and Q-learning play against each other. We will choose values which have shown to be effective in the previous learning sessions. In figure 14 the results are shown. From this we can see that MENACE and Q-learning perform very similar and are good opponents for each other.

We also make some interesting observations from the figure. In some training sessions Q-learning is performing a lot better than MENACE. This is due to the same effect as we saw in figure 7. So Q-learning is constantly playing a move which results in an empty box for MENACE thus making MENACE resign instantly. We also see that MENACE is consistently playing slightly better than Q-learning. On average however the algorithms have equal strength.

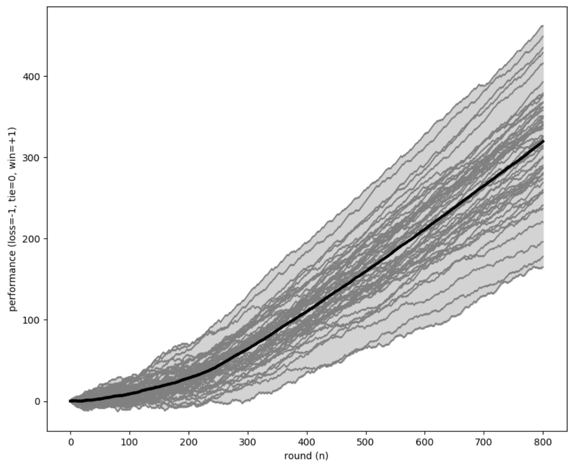

FIGURE 11. $Q(0.1, 0.1, 0.2, 200, 400, 1, 2, -1)$ against the random player with alternating turns. In total 50 training sessions.

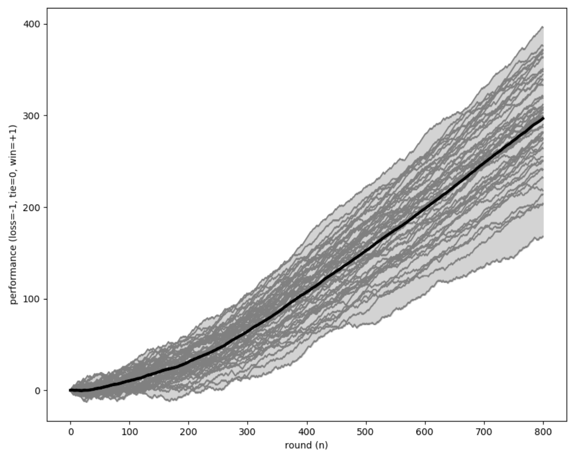

FIGURE 12. $Q(0.1, 0.9, 0.2, 200, 400, 1, 2, -1)$ against the random player with alternating turns. In total 50 training sessions.

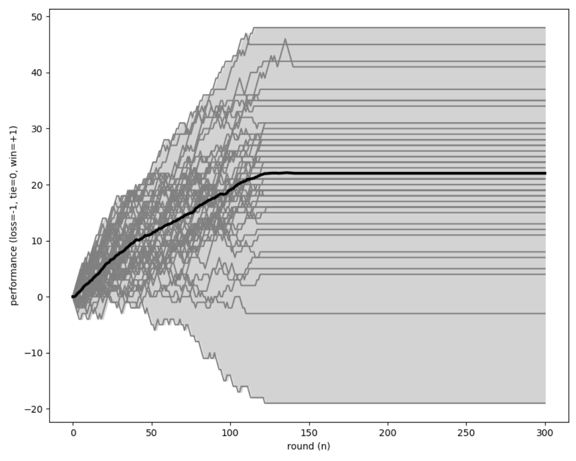

FIGURE 13. $Q_1(0.1, x, 0.2, 100, 200, 1, 2, -1)$ against $Q_2(0.1, x, 0.2, 100, 200, 1, 2, -1)$. In total 50 training sessions.

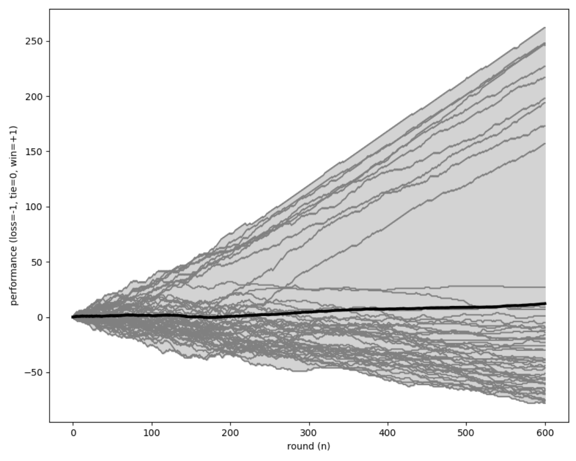

FIGURE 14. $Q(0.1, 0.9, 0.2, 200, 400, 1, 2, -1)$ against $M(1, 2, 1)$ with alternating turns. In total 50 training sessions.

5. CONCLUSION

The first observation I made is that MENACE and Q-learning both are quite adequate players for Noughts and Crosses. They both achieve perfect play relatively fast, within about 300 games at most.

Q-learning is also exceptionally strong against the perfect players, at some point it will always draw. MENACE is a little bit less good than Q-learning against the perfect players. It will almost always draw but sometimes it does make a incorrect move. This is due to how Q-learning and MENACE differ in how they choose their move. Q-learning always chooses the best move while MENACE chooses a move using probability. This means that MENACE is more likely to choose the correct move than the wrong move, but the wrong move can still be made.

We further saw that MENACE and Q-learning are good opponents for each other. They have about the same strength and learn at about the same rate if the most optimal variables are chosen.

Concerning the most optimal variables we have made some observations for both MENACE and Q-learning. With MENACE a big amount of initial beads per move per box is crucial for survival against a perfect player. However a small amount can be chosen to improve the speed at which MENACE will learn against a random player. We further saw that the reward for a win does not matter for MENACE and Q-learning when playing against a perfect player since they would never win.

Concerning Q-learning, a low learning rate will make Q-learning more stable than a high learning rate. We also saw that the exploration value can be quite low and that we can remove the exploration relatively fast, around 50-100 games. We further observed that the discount factor is irrelevant when Q-learning is always playing as either the first or second player. However it does improve the learning results a bit when the turn are alternated.

6. DISCUSSION

Here I will give some directions for future improvements and provide a small part on how the results correlate to what I expected.

6.1. Expectations

From what I read before I knew MENACE would be quite good at learning Noughts and Crosses. I also knew that it is possible for MENACE to die. Although I did not realise that this would also be possible when MENACE is always playing second, so I implemented this later in the process. I also expected MENACE to do better against the perfect players with a high amount of initial beads per move per box. Just as I expected it to learn faster with a low amount of initial beads. I also knew the reward for a win does not matter when playing against a perfect player.

For Q-learning I also had some expectations, but not all of those were correct. I thought for example that a high learning rate would make Q-learning learn faster which was not the case. A high learning rate just made the algorithm more unstable, but not better at learning. I further did not think about the fact that the discount factor is irrelevant when the turns are not alternated. However I knew I had to implement an exploration factor, which is not inherently part of the Q-learning algorithm, to prevent the algorithm from pursuing a local minimum. I also knew the exploration factor should be lowered the longer Q-learning was playing.

Lastly I expected Q-learning to be a lot better than MENACE since it is so much more complicated and newer however this was not really the case here. Q-learning is better when playing against a perfect player and Q-learning never dies in contrast to MENACE. However Q-learning plays equally good as MENACE in all other cases and when they are playing against each other.

6.2. Future

For the future there are some improvements which can be made. One important improvement concerns the reward system of MENACE. Currently when MENACE wins then all the moves will get an equal amount of extra beads. However this can be improved by giving the moves which were made later more beads than the moves which were made at the start. This makes sense because the later moves had a bigger impact on the result of the game. This would make the learning of MENACE more stable.

We can also improve the Q-learning algorithm. As said earlier if the players are not alternating turns then the discount factor is effectively useless. This could be resolved by looking further into the future and making a better prediction of future reward.

ACKNOWLEDGEMENTS

This research was undertaken as part of the project for the course Data Mining at the Radboud University.

REFERENCES

[1] Michie, D. (1961) Trial and error. *Science Survey*, **2**, 129–145. Harmondsworth: Penguin.

[2] Wikipedia contributors (2019). Q-learning — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=924370826`. [Online; accessed 24-November-2019].

[3] Wikipedia contributors (2019). Minimax — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Minimax&oldid=925373360`. [Online; accessed 25-November-2019].

[4] Wikipedia contributors (2019). Alpha–beta pruning — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=922064121`. [Online; accessed 25-November-2019].

[5] Wikipedia contributors (2019). Negamax — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Negamax&oldid=881512142`. [Online; accessed 25-November-2019].