Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play

Michiel van der Ree and Marco Wiering (*IEEE Member*)
Institute of Artificial Intelligence and Cognitive Engineering
Faculty of Mathematics and Natural Sciences
University of Groningen, The Netherlands

***Abstract*—This paper compares three strategies in using reinforcement learning algorithms to let an artificial agent learn to play the game of Othello. The three strategies that are compared are: Learning by self-play, learning from playing against a fixed opponent, and learning from playing against a fixed opponent while learning from the opponent's moves as well. These issues are considered for the algorithms Q-learning, Sarsa and TD-learning. These three reinforcement learning algorithms are combined with multi-layer perceptrons and trained and tested against three fixed opponents. It is found that the best strategy of learning differs per algorithm. Q-learning and Sarsa perform best when trained against the fixed opponent they are also tested against, whereas TD-learning performs best when trained through self-play. Surprisingly, Q-learning and Sarsa outperform TD-learning against the stronger fixed opponents, when all methods use their best strategy. Learning from the opponent's moves as well leads to worse results compared to learning only from the learning agent's own moves.**

I. Introduction

Many real-life decision problems are sequential in nature. People are often required to sacrifice an immediate pay-off for the benefit of a greater reward later on. Reinforcement learning (RL) is the field of research which concerns itself with enabling artificial agents to learn to make sequential decisions that maximize the overall reward [1], [2]. Because of their sequential nature, games are a popular application of reinforcement learning algorithms. The backgammon learning program TD-Gammon [3] showed the potential of reinforcement learning algorithms by achieving an expert level of play by learning from training games generated by self-play. Other RL applications to games include chess [4], checkers [5] and Go [6]. The game of Othello has also proven to be a useful testbed to examine the dynamics of machine learning methods such as evolutionary neural networks [7], n-tuple systems [8], and structured neural networks [9].

When using reinforcement learning to learn to play a game, an agent plays a large number of training games. In this research we compare different ways of learning from training games. Additionally, we look at how the level of play of the training opponent affects the final performance. These issues are investigated for three canonical reinforcement learning algorithms. TD-learning [10] and Q-learning [11] have both been applied to Othello before [9], [12]. Additionally, we compare the on-policy variant of Q-learning, Sarsa [13].

In using reinforcement learning to play Othello, we can use at least three different strategies: First, we can have a learning agent train against itself. Its evaluation function will become more and more accurate during training, and there will never be a large difference in level of play between the training agent and its opponent. A second strategy would be to train while playing against a player which is fixed, in the sense that its playing style does not change during training. The agent would learn from both its own moves and the moves its opponent makes. The skill levels of the non-learning players can vary. A third strategy consists of letting an agent train against a fixed opponent, but only have it learn from its own moves. This paper examines the differences between these three strategies. It attempts to answer the following research questions:

- How does the performance of each algorithm after learning through self-play compare to the performance after playing against a fixed opponent, whether paying attention to its opponent's moves or just its own?
- When each reinforcement learning algorithm is trained using its best strategy, which algorithm will perform best?
- How does the skill level of the fixed training opponent affect the final performance when the learning agent is tested against another opponent?

Earlier research considered similar issues for backgammon [14]. There, it was shown that learning from playing against an expert is the best strategy. However, in that paper only TD-learning and one strong fixed opponent were used. When learning from a fixed opponent's moves as well, an agent doubles the amount of training data it receives. However, it tries to learn a policy while half of the input it perceives was obtained by following a different policy. The problem may be that the learning agent cannot try out its own preferred moves to learn from, when the fixed opponent selects them. This research will show whether this doubling of training data is able to compensate for the inconsistency of policies. It is not our goal to develop the best Othello playing computer program, but we are interested in these research questions that also occur in other applications of RL.

In our experimental setup, three benchmark players will be used in both the train runs and the test runs. The results will therefore also show possible differences between the effect this

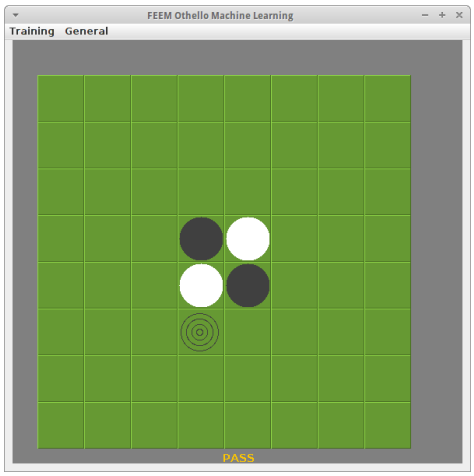


Figure 1. Screenshot of the used application showing the starting position of the game. The black circles indicate one of the possible moves for the current player (black).

similarity between training and testing will have on the test performance for each of the three algorithms.

Outline. In section II we shortly explain the game of Othello. In section III, we discuss the theory behind the used algorithms. Section IV describes the experiments that we performed and the results obtained. A conclusion will be presented in section V.

II. Othello

Othello is a two-player game played on a board of 8 by 8 squares. Figure 1 shows a screenshot of our application with the starting position of the game. The white and the black player place at alternate turns one disc at a time. A move is only valid if the newly placed disc causes one or more of the opponent's discs to become enclosed. The enclosed discs are then flipped, meaning that they change color. If and only if a player cannot capture any of the opponent's discs the player passes. When both players have to pass the game is ended. The player who has the most discs of his own color is declared winner, when the number of discs of each color are equal a draw is declared.

The best known Othello playing program is LOGISTELLO [15]. In 1997, it defeated the then world champion T. Murakami with a score of 6-0. The program was trained in several steps: First, logistic regression was used to map the features of the disc differential at the end of the game. Then, it used 13 different game stages and sparse linear regression to assign values to pattern configurations [16]. Its evaluation function was then trained on several millions of training positions to fit approximately 1.2 million weights [15].

III. Reinforcement Learning

In this section we give an introduction to reinforcement learning and sequential decision problems. In reinforcement learning, the learner is a decision making agent that takes actions in an environment and receives a reward (or penalty) for its actions in trying to solve a problem [1], [2]. After a set of trial-and-error runs it should learn the best policy, which is the sequence of actions that maximize the total reward.

We assume an underlying Markov decision process, which is formally defined by (1) A finite set of states $s \in S$; (2) A finite set of actions $a \in A$; (3) A transition function $T(s, a, s')$, specifying the probability of ending in state s' after taking action a in state s; (4) A reward function $R(s, a)$, providing the reward the agent will receive for executing action a in state s, where r_t denotes the reward obtained at time t; (5) A discount factor $0 \leq \gamma \leq 1$ which discounts later rewards compared to immediate rewards.

A. Value Functions

We want our agent to learn an optimal *policy* for mapping states to actions. The policy defines the action to be taken in any state $s : a = \pi(s)$. The value of a policy π, $V^\pi(s)$, is the expected cumulative reward that will be received when the agent follows the policy starting at state s. It is defined as:

$$V^\pi(s) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, \pi\right], \tag{1}$$

where $E[.]$ denotes the expectancy operator. The optimal policy is the one which has the largest state-value in all states.

Instead of learning values of states $V(s_t)$ we could also choose to work with values of state-action pairs $Q(s_t, a_t)$. $V(s_t)$ denotes how good it is for the agent to be in state s_t whereas $Q(s_t, a_t)$ denotes how good it is for the agent to perform action a_t in state s_t. The Q-value of such a state-action pair $\{s, a\}$ is given by:

$$Q^\pi(s, a) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, a_0 = a, \pi\right]. \tag{2}$$

B. Reinforcement Learning Algorithms

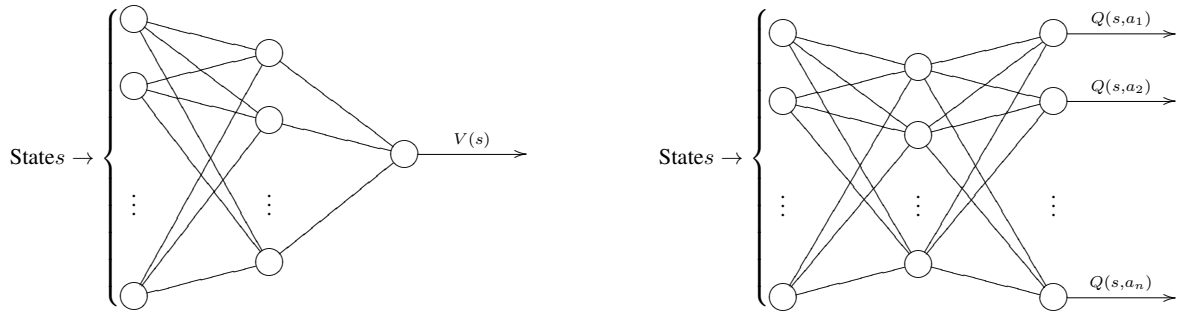
When playing against an opponent, the results of the agent's actions are not deterministic. After the agent has made its move, its opponent moves. In such a case, the Q-value of a certain state-action pair is given by:

$$Q(s_t, a_t) = E[r_t] + \gamma \sum_{s_{t+1}} T(s_t, a_t, s_{t+1}) \max_a Q(s_{t+1}, a) \tag{3}$$

Here, s_{t+1} is the state the agent encounters *after* its opponent has made his move. We cannot do a direct assignment in this case because for the same state and action, we may receive a different reward or move to different next states. What we *can* do is keep a running average. This is known as the *Q-learning algorithm* [11]:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r_t + \gamma \max_a \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t)) \tag{4}$$

where $0 < \alpha \leq 1$ is the learning rate. We can think of (4) as reducing the difference between the current Q value and the backed-up estimate. Such algorithms are called temporal difference algorithms [10]. Once the algorithm is finished, the



(a) TD network (b) Q-learning network

Figure 2. Topologies of function approximators. A TD-network (a) tries to approximate the value of the state presented at the input. A Q-learning network (b) tries to approximate the values of all the possible actions in the state presented at the input.

agent can use the value of state action pairs to select the action with the best expected outcome:

$$\pi(s) = \arg\max_a \hat{Q}(s,a) \tag{5}$$

If an agent would only follow the strategy it estimates to be optimal, it might never learn better strategies, because the action values can remain highest for the same actions in all different states. To circumvent this, an exploration strategy should be used. In ε-greedy exploration, there is a probability of ε that the agent executes a random action, and otherwise it selects the action with the highest state-action value. ε tends to be gradually decreased during training.

Sarsa, the on-policy variant of Q-learning, takes this exploration strategy into account. It differs from Q-learning in that it does not use the discounted Q-value of the subsequent state with the highest Q-value to estimate the Q-value of the current state. Instead, it uses the discounted Q-value of the state-action pair that occurs when using the exploration strategy:

$$\hat{Q}(s_t,a_t) \leftarrow \hat{Q}(s_t,a_t) + \alpha(r_t + \gamma\hat{Q}(s_{t+1},a_{t+1}) - \hat{Q}(s_t,a_t)) \tag{6}$$

where a_{t+1} is the action prescribed by the exploration strategy.

The idea of temporal differences can also be used to learn $V(s)$ values, instead of $Q(s,a)$. TD learning (or TD(0) [10]) uses the following update rule to update a state value:

$$V(s_t) \leftarrow V(s_t) + \alpha\left(r_t + \gamma V(s_{t+1}) - V(s_t)\right) \tag{7}$$

C. Function Approximators

In problems of modest complexity, it might be feasible to actually store the values of all states or state-action pairs in lookup tables. However, Othello's state space size is approximately 10^{28} [12]. This is problematic for at least two reasons. First of all, the space complexity of the problem is much too large to be stored. Furthermore, after training our agent it might be asked to evaluate states or state-action pairs which it has not encountered during training and it would have no clue how to do so. Using a lookup table would cripple the agent's ability to generalize to unseen input patterns.

For these two reasons, we instead train multi-layer perceptrons to estimate the $V(s)$ and $Q(s,a)$ values. During the learning process, the neural network learns a mapping from state descriptions to either $V(s)$ or $Q(s,a)$ values. This is done by computing a 'target' value according to (4) in the case of Q-learning or (7) in the case of TD-learning. The learning rate α in these functions is set to 1, since we already have the learning rate of the neural network to control the effect training examples have on estimations of $V(s)$ or $Q(s,a)$. This means that (4) and (6) respectively simplify to

$$\hat{Q}^{\text{new}}(s_t,a_t) \leftarrow r_t + \gamma \max_a \hat{Q}(s_{t+1},a) \tag{8}$$

and

$$\hat{Q}^{\text{new}}(s_t,a_t) \leftarrow r_t + \gamma\hat{Q}(s_{t+1},a_{t+1}). \tag{9}$$

Similarly, (7) simplifies to

$$V^{\text{new}}(s_t) \leftarrow r_t + \gamma V(s_{t+1}). \tag{10}$$

In the case of TD-learning, for example, we use $(s_t, V^{\text{new}}(s_t))$ as training example for the neural network trained with the backpropagation algorithm. A Q-learning or Sarsa network consists of one or more input units to represent a state. The output consists of as many units as there are actions that can be chosen. A TD-learning network also has one or more input units to represent a state. It has a single output approximating the value of the state given as input. Figure 2 illustrates the structure of both networks.

D. Application to Othello

In implementing all three learning algorithms in our Othello framework, there is one important factor to account for: The fact that we have to wait for our opponent's move before we can learn either a $V(s)$ or a $Q(s,a)$ value. Therefore, we learn the value of the *previous* state or state-action pair at the beginning of each turn – that is, before a move is performed. Every turn except the first, our Q-learning agent goes through the following steps:

1) Observe the current state s_t
2) For all possible actions a'_t in s_t use NN to compute $\hat{Q}(s_t,a'_t)$
3) Select an action a_t using a policy π
4) According to (8) compute the target value of the previous state-action pair $\hat{Q}^{\text{new}}(s_{t-1},a_{t-1})$

5) Use NN to compute the current estimate of the value of the previous state-action pair $\hat{Q}(s_{t-1}, a_{t-1})$
6) Adjust the NN by backpropating the error $\hat{Q}^{\text{new}}(s_{t-1}, a_{t-1}) - \hat{Q}(s_{t-1}, a_{t-1})$
7) $s_{t-1} \leftarrow s_t$, $a_{t-1} \leftarrow a_t$
8) Execute action a_t

Note that only the output unit belonging to the previously executed action is adapted. For all other output units, the error is set to 0. The Sarsa implementation is very similar, except that in step 4 it uses (9) to compute the target value of the previous state-action pair instead of (8).

In online TD-learning we are learning values of *afterstates*, that is: the state directly following the execution of an action, before the opponent has made its move. During playing, the agent can then evaluate all accessible afterstates and choose the one with the highest $V(s^a)$. Each turn except the first, our TD-agent performs the following steps:

1) Observe the current state s_t
2) For all afterstates s'_t reachable from s_t use NN to compute $V(s'_t)$
3) Select an action leading to afterstate s^a_t using a policy π
4) According to (10) compute the target value of the previous afterstate $V^{\text{new}}(s^a_{t-1})$
5) Use NN to compute the current value of the previous afterstate $V(s^a_{t-1})$
6) Adjust the NN by backpropating the error $V^{\text{new}}(s^a_{t-1}) - V(s^a_{t-1})$
7) $s^a_{t-1} \leftarrow s^a_t$
8) Execute action resulting in afterstate s^a_t

E. Learning from Self-Play and Against an Opponent

We compare three strategies by which an agent can learn from playing training games: playing against itself; learning from playing against a fixed opponent using both its own moves and the opponent's moves, and learning from playing against a fixed opponent using only its own moves.

1) Learning from Self-Play: When learning from self-play, we have both agents share the same neural network which is used for estimating the $Q(s, a)$ and $V(s)$ values. In this case, both agents use the algorithm described in subsection III-D, adjusting the weights of the same neural network.

2) Learning from Both Own and Opponent's Moves: When an agent learns from both its own moves and its opponent's moves, it still learns from its own moves according to the algorithms described in subsection III-D. In addition to that, it also keeps track of its opponent's moves and previously visited (after-)states. Once an opponent has chosen an action a_t in state s_t, the Q-learning and Sarsa agent will:

1) Compute the target value of the opponent's previous state-action pair $\hat{Q}^{\text{new}}(s_{t-1}, a_{t-1})$ according to (8) for Q-learning or (9) for Sarsa
2) Use the NN to compute the current estimate of the value of the opponent's previous state action pair $\hat{Q}(s_{t-1}, a_{t-1})$
3) Adjust the NN by backpropating the difference between the target and the estimate

Similarly, when the TD-agent learns from its opponent it will do the following once an opponent has reached an afterstate s^a_t:

1) According to (10) compute the target value of the opponents previous afterstate $V^{\text{new}}(s^a_{t-1})$
2) Use NN to compute the current value of the opponents previous afterstate $V(s^a_{t-1})$
3) Adjust the NN by backpropating the difference between the target and the estimate

3) Learning from Its Own Moves: When an agent plays against a fixed opponent and only learns from its own moves, it simply follows the algorithm described in subsection III-D, without keeping track of the moves its opponent made and the (after-)states its opponent visited.

IV. Experiments and Results

In training our learning agents, we use feedforward multi-layer perceptrons with one hidden layer consisting of 50 hidden nodes as function approximators. All parameters, including the number of hidden units and the learning rates, were optimized during a number of preliminary experiments. A sigmoid function:

$$f(a) = \frac{1}{1 + e^{-a}} \tag{11}$$

is used on both the hidden and the output layer. The weights of the neural networks are randomly initialized to values between -0.5 and 0.5. States are represented by an input vector of 64 nodes, each corresponding to a square on the Othello board. Values corresponding to squares are 1 when the square is taken by the learning agent in question, -1 when it is taken by its opponent and 0 when it is empty. The reward associated with a terminal state is 1 for a win, 0 for a loss and 0.5 for a draw. The discount factor γ is set to 1.0. The probability of exploration ε is initialized to 0.1 and linearly decreases to 0 over the course of all training episodes. The learning rate for the neural network is set to 0.01 for Q-learning and Sarsa, and for TD-learning a value of 0.001 is used.

100	-25	10	5	5	10	-25	-100
-25	-25	2	2	2	2	-25	-25
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-25	-25	2	2	2	2	-25	-25
100	-25	10	5	5	10	-25	100

(a)

80	-26	24	-1	-5	28	-18	76
-23	-39	-18	-9	-6	-8	-39	-1
46	-16	4	1	-3	6	-20	52
-13	-5	2	-1	4	3	-12	-2
-5	-6	1	-2	-3	0	-9	-5
48	-13	12	5	0	5	-24	41
-27	-53	-11	-1	-11	-16	-58	-15
87	-25	27	-1	5	36	-3	100

(b)

Figure 3. Positional values used by player HEUR (a) and player BENCH (b, trained using co-evolution [17]).

A. Fixed Players

We created three fixed players: one random player RAND and two positional players, HEUR and BENCH. These players are both used as fixed opponents and benchmark players. The random player always takes a random move based on the available actions. The positional players have a table attributing values to all squares of the game board. They use the following evaluation function:

$$V = \sum_{i=1}^{64} c_i w_i \tag{12}$$

where c_i is 1 is the square i is occupied by the player's own disc, -1 when it is occupied by an opponent's disc and 0 when it is unoccupied, and w_i is the positional value of a square i. The two positional players differ in the weights w_i they attribute to squares. Player HEUR uses weights used in multiple other Othello researches [18], [17], [9]. Player BENCH uses an evaluation function created using co-evolution [17] and has been used as a benchmark player before as well [9]. The weights used by HEUR and BENCH are shown in figure 3.

The positional players use (12) to evaluate the state directly following an own possible move, i.e. before the opponent has made a move in response. They choose the action which results in the afterstate with the highest value.

Table I
PERFORMANCES OF THE FIXED STRATEGIES WHEN PLAYING AGAINST EACH OTHER. THE PERFORMANCES OF THE GAMES INVOLVING PLAYER RAND ARE THE AVERAGES OF 472.000 GAMES (1.000 GAMES FROM EACH OF THE 472 DIFFERENT STARTING POSITIONS).

HEUR - BENCH	BENCH - RAND	RAND - HEUR
0.55 - 0.45	0.80 - 0.20	0.17 - 0.83

B. Testing the Algorithms

To gain a good understanding of the performances of both the learning and the fixed players, we let them play multiple games, both players playing black and white. All players except RAND have a deterministic strategy during testing. To prevent having one player win all training games, we initialize the board as one of 236 possible starting positions after four turns[1]. During both training and testing, we cycle through all the possible positions, ensuring that all positions are used the same number of times. Each position is used twice: the agent plays both as white and black. Table I shows the average performance per game of the fixed strategies when tested against each other in this way. We are interested in whether the relative performances might be reflected in the learning player's performance when training against the three fixed players.

[1]In other literature, 244 possible board configurations after four turns are mentioned. We found there to be 244 different sequences of legal moves from the starting board to the fifth turn, but that they result in 236 unique positions.

Table II
PERFORMANCES OF THE LEARNING ALGORITHMS WHEN TESTED VERSUS PLAYER BENCH. EACH COLUMN SHOWS THE PERFORMANCE IN THE TEST SESSION WHERE THE LEARNING PLAYER PLAYED BEST, AVERAGED OVER A TOTAL OF TEN EXPERIMENTS. THE STANDARD ERROR ($\hat{\sigma}/\sqrt{n}$) IS SHOWN AS WELL.

Train vs.	Q-learning	Sarsa	TD-Learning
BENCH	$\mathbf{0.871 \pm 0.009}$	$\mathbf{0.859 \pm 0.006}$	0.700 ± 0.007
BENCH-LRN	0.780 ± 0.008	0.816 ± 0.006	0.628 ± 0.009
Itself	0.721 ± 0.011	0.699 ± 0.011	$\mathbf{0.723 \pm 0.017}$
HEUR	0.582 ± 0.008	0.574 ± 0.018	0.522 ± 0.008
HEUR-LRN	0.563 ± 0.008	0.427 ± 0.015	0.355 ± 0.010
RAND	0.330 ± 0.010	0.307 ± 0.009	0.356 ± 0.011
RAND-LRN	0.418 ± 0.018	0.300 ± 0.012	0.332 ± 0.008

C. Comparison

We use the fixed players both to train the algorithms and to test them. In the experiments in which players HEUR and BENCH were used as opponents in the test games a total of 2,000,000 games were played during training. After each 20,000 games of training, the algorithms played 472 games versus respectively BENCH or HEUR without exploration. Tables II and III show the averages of the best performances of each algorithm when testing against players BENCH and HEUR after having trained against the various opponents through the different strategies: Itself, HEUR, HEUR when learning from its opponent's moves (HEUR-LRN), BENCH, BENCH when learning from its opponent's moves (BENCH-LRN), RAND and RAND when learning from its opponent's moves (RAND-LRN).

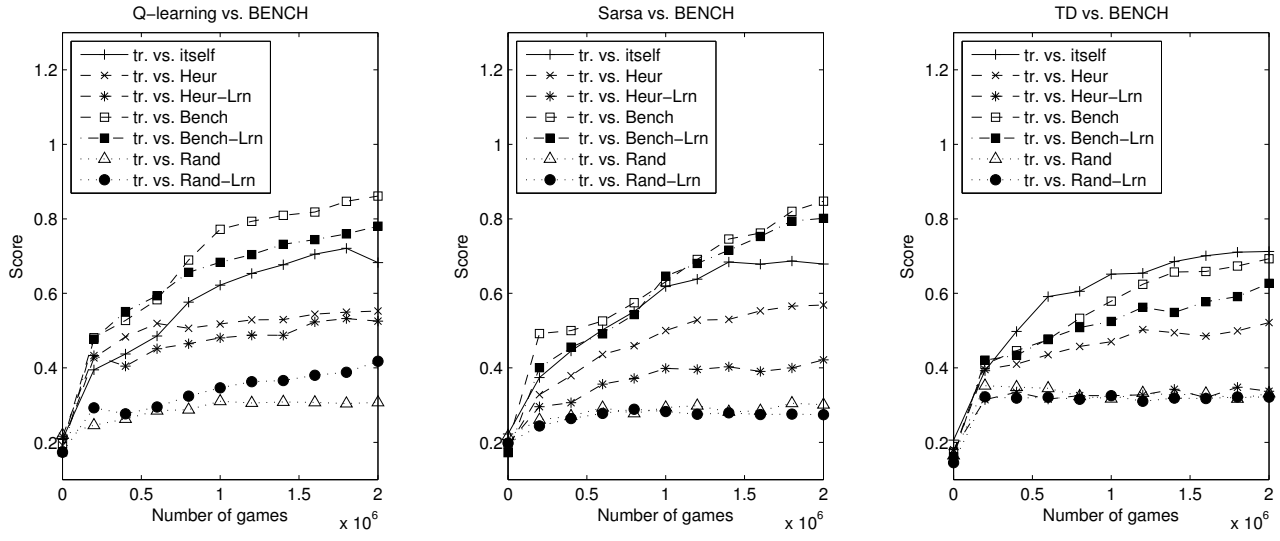
Table III
PERFORMANCES OF THE LEARNING ALGORITHMS WHEN TESTED VERSUS PLAYER HEUR. EACH COLUMN SHOWS THE PERFORMANCE IN THE TEST SESSION WHERE THE LEARNING PLAYER PLAYED BEST, AVERAGED OVER A TOTAL OF TEN EXPERIMENTS. THE STANDARD ERROR ($\hat{\sigma}/\sqrt{n}$) IS SHOWN AS WELL.

Train vs.	Q-learning	Sarsa	TD-Learning
HEUR	$\mathbf{0.810 \pm 0.009}$	$\mathbf{0.809 \pm 0.005}$	$\mathbf{0.775 \pm 0.005}$
HEUR-LRN	0.651 ± 0.006	0.728 ± 0.013	0.666 ± 0.007
Itself	0.641 ± 0.016	0.631 ± 0.015	0.767 ± 0.005
BENCH-LRN	0.476 ± 0.012	0.361 ± 0.011	0.725 ± 0.009
BENCH	0.397 ± 0.016	0.440 ± 0.012	0.708 ± 0.009
RAND-LRN	0.356 ± 0.014	0.498 ± 0.009	0.610 ± 0.010
RAND	0.426 ± 0.007	0.428 ± 0.016	0.644 ± 0.015

Table IV
PERFORMANCES OF THE LEARNING ALGORITHMS WHEN TESTED VERSUS PLAYER RAND. EACH COLUMN SHOWS THE PERFORMANCE IN THE TEST SESSION WHERE THE LEARNING PLAYER PLAYED BEST, AVERAGED OVER A TOTAL OF TEN EXPERIMENTS. THE STANDARD ERROR ($\hat{\sigma}/\sqrt{n}$) IS SHOWN AS WELL.

Train vs.	Q-learning	Sarsa	TD-Learning
Itself	$\mathbf{0.949 \pm 0.003}$	$\mathbf{0.946 \pm 0.002}$	$\mathbf{0.975 \pm 0.002}$
RAND	0.893 ± 0.006	0.906 ± 0.005	0.928 ± 0.003
BENCH-LRN	0.893 ± 0.004	0.896 ± 0.003	0.924 ± 0.007
RAND-LRN	0.892 ± 0.007	0.885 ± 0.004	0.917 ± 0.003
HEUR-LRN	0.914 ± 0.004	0.837 ± 0.007	0.814 ± 0.008
HEUR	0.850 ± 0.007	0.827 ± 0.007	0.912 ± 0.004
BENCH	0.792 ± 0.017	0.783 ± 0.018	0.879 ± 0.007

For each test session, the results were averaged over a total of ten experiments. The tables show the averaged results in the session in which the algorithms, on average, performed best.



(a) (b) (c)

Figure 4. Average performance of the algorithms over ten experiments. With 2,000,000 of training games against the various opponents and testing Q-learning, Sarsa and TD-learning versus player BENCH (a, b and c respectively).

Figures 4 and 5 show how the performance develops during training when tested versus players BENCH and HEUR. The performances in the figures are a bit lower than in the tables, because in the tables the best performance during an epoch is used to compute the final results.

In the experiments in which the algorithms are tested versus player RAND, a total of 500,000 training games were played. Table IV shows the best performance when training against each of the various opponents through the different strategies. Figure 6 shows how the performance develops during training when testing versus player RAND.

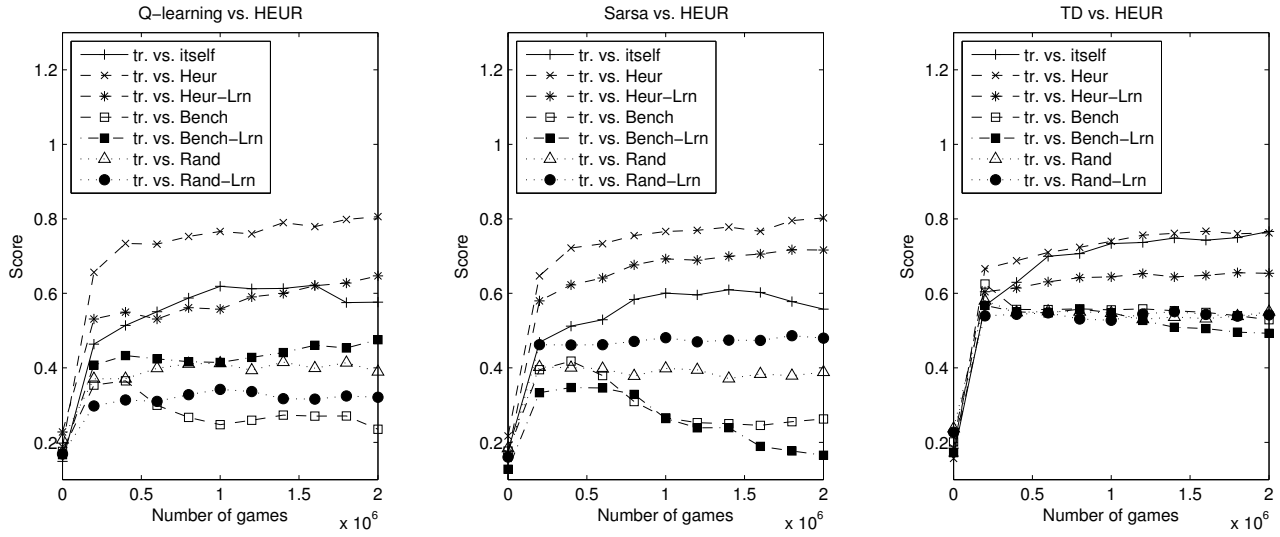
D. Discussion

These results allow for the following observations:

- **Mixed policies** There is not a clear benefit to paying attention to the opponent's moves when learning against a fixed player. Tables II, III and IV seem to indicate that the doubling of perceived training moves does not improve performance as much as getting input from different policies decreases it.
- **Generalization** Q-learning and Sarsa perform best when having trained with the same player against which they are tested. When training against that player, the performance is best when the learning player does not pay attention to its opponent's moves. For both Q-learning and Sarsa, training against itself comes in at a third place in the experiments where the algorithms are tested versus HEUR and BENCH. For TD-learning, however, the performance when training against itself is similar or even better than the performance after training against the same player used in testing. This seems to indicate that the TD-learner achieves a higher level of generalization. This is due to the fact that the TD-learner learns values of states while the other two algorithms learn values of actions in states.
- **Symmetry** The TD-learner achieves a low performance against BENCH when having trained against HEUR-LRN, RAND and RAND-LRN. However, the results of the TD-learner when tested against HEUR lack a similar result. We speculate that this can be attributed to the lack of symmetry in BENCH's positional values.

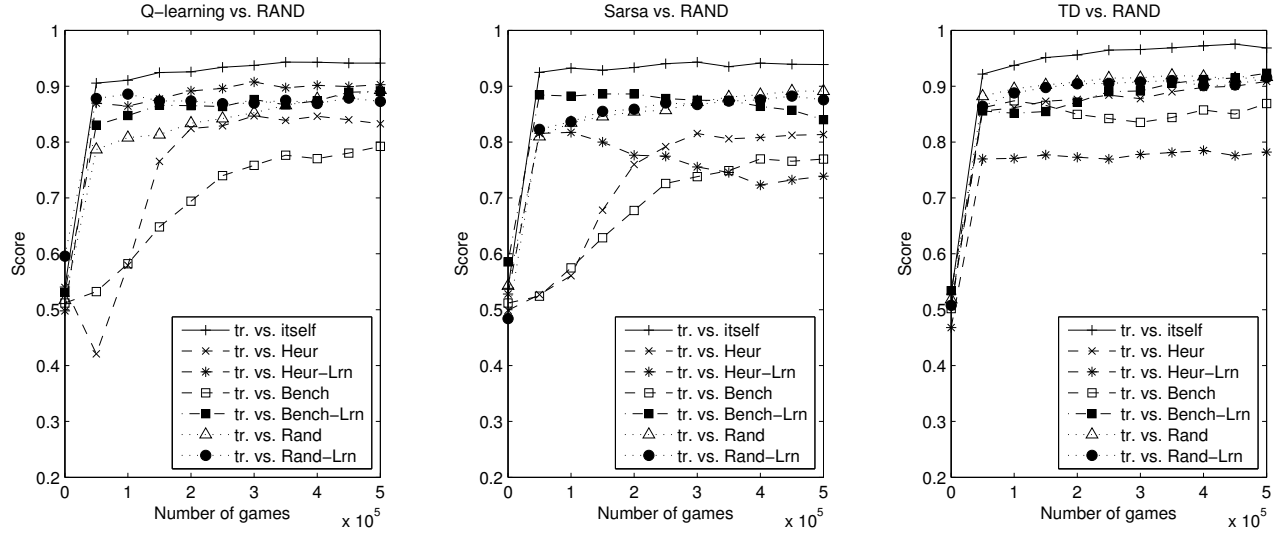
Using our results, we can now return to the research questions posed in the introduction:

- **Question** How does the performance of each algorithm after learning through self-play compare to the performance after playing against a fixed opponent, whether paying attention to its opponent's moves or just its own?
  Answer Q-learning and Sarsa learn best when they train against the same opponent against which they are tested. TD-learning seems to learn best when training against itself. None of the algorithms benefit from paying attention to its opponent's moves when training against a fixed strategy. We believe this is because the RL agent is not free to choose its own moves when the opponent selects a move, leading to a biased policy.
- **Question** When each reinforcement learning algorithm is trained using its best strategy, which algorithm will perform best?
  Answer When Q-learning and Sarsa train against BENCH and HEUR without learning from their opponent's moves while tested against the same players, they clearly outperform TD after it has trained against itself. This is a surprising result, since we expected TD-learning to perform better. However, if we compare the performance for each of the three algorithms after training against itself, TD significantly outperforms Q-learning and Sarsa when



(a) (b) (c)

Figure 5. Average performance of the algorithms over ten experiments. With 2,000,000 of training games against the various opponents and testing Q-learning, Sarsa and TD-learning versus player HEUR (a, b and c respectively).



(a) (b) (c)

Figure 6. Average performance of the algorithms over ten experiments. With 500,000 games of training against the various opponents and testing Q-learning, Sarsa and TD-learning versus player RAND (a, b and c respectively).

tested against HEUR and RAND. When tested against BENCH after training against itself, the difference between TD-learning and Q-learning is insignificant. The obtained performances of Q-learning and Sarsa are very similar.

- **Question** How does the skill level of the fixed training opponent affect the final performance when the learning agent is tested against another fixed opponent?
  Answer From table I we see that player HEUR performs better against RAND than BENCH. This is also reflected in the performances of the algorithms versus RAND after having trained with HEUR and BENCH respectively. From table I we see as well that HEUR has a better performance than BENCH when the two players play against each other. This difference in performance also seems to be partly reflected in our results: When Q-learning and Sarsa train against player HEUR they obtain a higher performance when tested against BENCH than vice versa. However, we don't find a similar result for TD-learning. That might be attributed to the fact that BENCH's weights values are not symmetric and therefore BENCH might pose a greater challenge to TD-learning than to Q-learning and Sarsa. We believe that BENCH can be better exploited

using different action networks, as used by Q-learning and Sarsa, since particular action sequences follow other action sequences in a more predictable way when playing against BENCH. Because TD-learning only uses one state network, it cannot easily exploit particular action sequences.

V. Conclusion

In this paper we have compared three strategies in using reinforcement learning algorithms to learn to play Othello: learning by self-play, learning by playing against a fixed opponent and learning by playing against a fixed opponent while learning from the opponent's moves as well. We found that it differs per algorithm what the best strategy is to train: Q-learning and Sarsa obtain the highest performance when training against the same opponent as which they are tested against (while *not* learning from the opponent's moves) while TD-learning learns best from self-play. Differences in the level of the training opponent seem to be reflected in the eventual performance of the training algorithms.

Future work might take a closer look at the influence of the training opponent's play style on the learned play style of the reinforcement learning agent. In our research, the differences in eventual performance were only analyzed in terms of a score. It would be interesting to experiment with fixed opponents with more diverse strategies and analyze the way these strategies influence the eventual play style of the learning agent in a more qualitative fashion.

References

[1] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. The MIT press, Cambridge MA, A Bradford Book, 1998.

[2] M. Wiering and M. van Ottelo, Eds., *Reinforcement Learning: State-of-the-art*. Springer, 2012.

[3] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, pp. 58–68, 1995.

[4] S. Thrun, "Learning to play the game of chess," *Advances in Neural Information Processing Systems*, vol. 7, 1995.

[5] J. Schaeffer, M. Hlynka, and V. Jussila, "Temporal difference learning applied to a high-performance game-playing program," in *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*. Morgan Kaufmann Publishers Inc., 2001, pp. 529–534.

[6] N. Schraudolph, P. Dayan, and T. Sejnowski, "Temporal difference learning of position evaluation in the game of go," *Advances in Neural Information Processing Systems*, pp. 817–817, 1994.

[7] D. Moriarty and R. Miikkulainen, "Discovering complex othello strategies through evolutionary neural networks," *Connection Science*, vol. 7, no. 3, pp. 195–210, 1995.

[8] S. Lucas, "Learning to play othello with n-tuple systems," *Australian Journal of Intelligent Information Processing*, vol. 4, pp. 1–20, 2008.

[9] S. van den Dries and M. Wiering, "Neural-fitted td-learning for playing othello with structured neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 11, pp. 1701–1713, 2012.

[10] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[11] C. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[12] N. van Eck and M. van Wezel, "Application of reinforcement learning to the game of othello," *Computers & Operations Research*, vol. 35, no. 6, pp. 1999–2017, 2008.

[13] G. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. Technical Report, University of Cambridge, Department of Engineering, 1994.

[14] M. Wiering, "Self-play and using an expert to learn to play backgammon with temporal difference learning," *Journal of Intelligent Learning Systems and Applications*, vol. 2, no. 2, pp. 57–68, 2010.

[15] M. Buro, "The evolution of strong othello programs," in *Entertainment Computing - Technology and Applications*, R. Nakatsu and J. Hoshino, Eds. Kluwer, 2003, pp. 81–88.

[16] ——, "Statistical feature combination for the evaluation of game positions," *Journal of Artificial Intelligence Research*, vol. 3, pp. 373–382, 1995.

[17] S. Lucas and T. Runarsson, "Temporal difference learning versus co-evolution for acquiring othello position evaluation," in *Computational Intelligence and Games, 2006 IEEE Symposium on*, 2006, pp. 52–59.

[18] T. Yoshioka and S. Ishii, "Strategy acquisition for the game othello based on reinforcement learning," *IEICE Transactions on Information and Systems*, vol. 82, no. 12, pp. 1618–1626, 1999.