Double Q–learning Agent for Othello Board Game

Thamarai Selvi Somasundaram
Department of Computer Technology
Madras Institute of Technology
Chennai, India
stselvi@annauniv.edu

Karthikeyan Panneerselvam
Department of Computer Technology
Madras Institute of Technology
Chennai, India
karthikn2789@gmail.com

Tarun Bhuthapuri
Chennai, India
tarun.bhuthapuri@protonmail.com

Harini Mahadevan
Bangalore, India
harini2008.katrina@gmail.com

Ashik Jose
Bangalore, India
ashikjose3@gmail.com

***Abstract*—This paper presents the first application of the Double Q–learning algorithm to the game of Othello. Reinforcement learning has previously been successfully applied to Othello using the canonical reinforcement learning algorithms, Q–learning and TD–learning. However, the algorithms suffer from considerable drawbacks. Q–learning frequently tends to be overoptimistic during evaluation, while TD–learning can get stuck in local minima. To overcome the disadvantages of the existing work, we propose using a Double Q–learning agent to play Othello and prove that it performs better than the existing learning agents. In addition to developing and implementing the Double Q–learning agent, we implement the Q–learning and TD–learning agents. The agents are trained and tested against two fixed opponents: a random player and a heuristic player. The performance of the Double Q–learning agent is compared with performance of the existing learning agents. The Double Q–learning agent outperforms them, although it takes longer, on average, to make each move. Further, we show that the Double Q–learning agent performs at its best with two hidden layers using the tanh function.**

Keywords—Double Q-learning, Othello, reinforcement learning

I. Introduction

Playing a game requires making sequential decisions. Decisions that appear to benefit a player immediately might ultimately result in the player losing the game. A player has to consider not just the immediate impact of a decision, but also how the decision will play out further into the future. Playing to win entails maximizing the overall payoff and, hence, in many cases, an immediate high reward might have to be sacrificed in order to achieve an overall high reward.

Reinforcement learning (RL) is a field of statistics which aims to make artificial decision-making agents successfully learn a policy to attain a complex goal. RL agents make sequential decisions which help maximize an overall reward. The problems solved by reinforcement learning can be modeled by a Markov Decision Process (MDP), which is stochastic and has the property that the decision at each state is made considering only the current state, without relying on earlier states or actions.

Games, with their step-by-step play, are well-suited for developing reinforcement learning agents. Reinforcement learning has been successfully applied to several games like poker [1], chess [2], checkers [3], and Atari games [4]. A long-standing popular example has been TD–Gammon [5], which displays a highly proficient level of play against humans in the game of backgammon when trained by play against another instance of itself. More recently, the artificial intelligence lab DeepMind created the learning agent AlphaGo [6] for the highly complex game of Go, which beat the world champion Lee Sedol in 2016 at a five-match series by a 4-1 margin. DeepMind followed up on that achievement by creating an improved version, called AlphaGo Zero [7], which beat AlphaGo in a hundred-game series by a 100-0 margin.

Reinforcement learning has also been successfully applied to the game of Othello [8] [9] [10]. Existing literature includes learning agents devised using multiple methods of reinforcement learning such as n-tuple systems [11], evolutionary neural networks [12], and structured neural networks [13]. A well-known Othello game agent is LOGISTELLO, which, in 1997, beat the world champion at the time, Takeshi Murakami, in a six-game series, 6-0.

While being canonical algorithms, Q–learning and TD–learning have their limitations. Q–learning frequently overestimates values, which tends to hamper performance. As for TD–learning, though it does not suffer from the bias of overestimation that affects Q–learning, it requires a defined policy to learn. To address the drawbacks of Q–learning, Double Q–learning was proposed [14]. It is an off-policy algorithm which reduces the overestimations caused by the standard Q–learning algorithm and provides a stable and reliable learning method. In application to roulette, grid world [14], and Atari games [15], it is shown to have faster convergence with a significant reduction in error rate.

We aim to develop and implement a Double Q–learning agent for the game of Othello, which uses two neural networks operating together to calculate the action values instead of a single neural network. We do not aim to create an Othello agent that achieves performance better than every other existing Othello agent. Our interest lies mainly in the practical implications of the application of the theoretically promising, and relatively new, Double Q–learning algorithm to the game of Othello.

The Q–learning, the TD–learning, and the Double Q–learning agents are implemented, trained and tested against two fixed opponents: a player which makes each move randomly, and a heuristic player which makes its moves based on a table of positional weights. We compare the performance of the three agents and establish that the Double Q–learning agent shows the best performance among them. In addition, we explore the usage of different activation functions, trying to identify which works best for our Othello playing agent. We also strive to address the question of whether the addition of another hidden layer improves performance.

The rest of the paper is organized as follows. Section 2 briefly describes the game of Othello. Section 3 gives an introduction to reinforcement learning and explains the existing algorithms used. It also describes the Double Q–learning algorithm and the theory behind the algorithm. Section 4 describes the Othello playing agents used in the experiments. Section 5 details the experiments done and the results obtained. Section 6 concludes the paper, and discusses possible future work.

II. The Game of Othello

Othello is a two-player game on an 8×8 board. The game is played using sixty-four identical disks, all of which are colored black on one side and white on the other side. Each player is given a color, black or white. The assigned color represents the player for the duration of the game. The two players take turns placing disks on the board with their assigned color facing up.

The game begins with the discs placed in a fixed initial position, shown in Fig. 1. The player with black makes the first move, followed by the player with white.

For the first move of the game, a black disk should be placed on a cell in the board such that there is at least one straight line (horizontal, vertical, or diagonal) between the new disk and a black disk already on the board, with one or more white disks in between, i.e. the newly placed black disk and one that is already on the board should entrap one or more white disks in between them. The entrapped white disk, or disks, immediately flip to black.

When the player with white plays, the same process is followed with the colors reversed. Thus, during a game, any disk of the opponent that is entrapped between the currently placed disk and a previously placed disk by the player in any direction is flipped to the color of the current player.

A player passes if there is no legal move available. This happens when no new disk can be placed in such a way that at least one of the opponent’s disks is entrapped. However, if a legal move is possible, the player cannot pass.

The game has the following terminal cases: when the last empty square is filled, or when neither of the players has any legal move left.

The objective for the player in this game is to have more disks on the board than the opponent when the game ends, in which case the player is the winner. If the game reaches a terminal state with an equal number of black and white disks on the board, it ends in a draw.

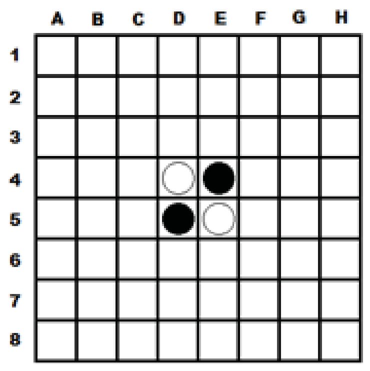

Fig. 1. Othello Starting Position

III. Reinforcement Learning Algorithms

In reinforcement learning, a decision-making agent learns by repeatedly interacting with the environment till the end of an episode. The environment, which in this case is the game space, is defined in terms of states. At any instant in time, the agent observes a particular state in the environment. The agent learns by taking actions, which leads to a new state in the environment. In addition to making a state transition, the agent also obtains reinforcement from the environment based on the action taken. The reinforcement can be positive (a reward) or negative (a penalty).

Reinforcement learning problems are defined by a Markov Decision Process (MDP), which is formally defined by (1) A finite set of states s ϵ S; (2) A finite set of actions a ϵ A; (3) A transition function T(s, a, s’), which gives the probability of ending in state s’ after taking action a in state s, (4) A reward function R(s, a), which provides the reward that the agent receives for executing action a in state s; (5) A discount factor $0 \leq \gamma \leq 1$ which discounts later rewards compared to immediate rewards.

MDPs have the property that the state transitions and rewards depend solely on the current state without considering past states and actions. Thus, the reinforcement learning agent learns and takes its actions solely based on the current state in the environment.

The aim of the agent is to learn an optimal policy to play the game, learning by trial and error. The policy must result in the agent performing a sequence of actions that maximizes the overall reward. As the aim is long-term victory as opposed to short-term gratification, the agent might have to compromise on immediate rewards in order to achieve an overall maximized reward.

Formally, we can define the goal of reinforcement learning as follows: the agent should learn an optimal policy which maps states to actions, Π:s→a. This policy is used to choose the best action for a state. The value of the policy is the expected value of the cumulative reward achieved by following the policy π from an initial state s and is given by,

$$V_{\pi}(s) \to E\left[\sum_{i=0}^{\infty} \gamma^{i} r_{i}\right] \tag{1}$$

r_i is the reward obtained upon taking an action in state s using the policy Π, and $\gamma \epsilon [0, 1]$ is the discount factor which determines the importance of the future rewards as opposed to immediate rewards. A value of $\gamma = 0$ results in consideration of only the immediate reward, while a value of $\gamma = 1$ will make it strive for a long-term high reward. States are observed from interaction with the environment. Taking an action from the current state transitions the game to a new state.

A. Q–learning

Q–learning is a widely-used algorithm in reinforcement learning. It is model-free. It does not require a model of the environment to learn and play. It uses a function called the Q-function to learn an optimal policy. The algorithm takes a (state, action) pair as input, and returns as output an updated Q–value for the pair. The Q–value represents how good the action would be if it were taken for the current state. The algorithm aims to obtain the maximum Q–value for the optimal action of a given state. The Q–value in a state s is given by,

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + (\gamma * \max(Q(s',a')))) \quad (2)$$

where,

s	→ Current state
a	→ Action selected for current state
α	→ Learning rate
Q()	→ Q-value function for (state,action) pair
r	→ Reward
γ	→ Discount factor
s'	→ Next state
a'	→ Any action

For any given state, the agent selects the maximum among the values corresponding to the actions. The action corresponding to this value is executed. The Q–value for the chosen (state, action) pair is updated according to (2). This results in a highly optimistic course of action in which the agent assumes that the maximum value denotes the optimal action. While this works in many scenarios, it need not always be true. When a suboptimal step ultimately leads to success, it gets rewarded, leaving the suboptimal path as an explored path which has been rewarded, even though the optimal path has not yet been explored. This preference towards following the highest existing action value causes a bias towards exploiting existing knowledge over exploring the unexplored portion of the state space, which could potentially contain the optimal path. This over-optimistic evaluation tends to adversely affect results, leading to the algorithm converging more slowly. At its slowest, convergence in Q–learning takes exponential time [14].

B. Temporal Difference (TD) learning

TD–learning is a method of reinforcement learning which learns after each step. It does not wait for the final outcome, and can learn from incomplete sequences. It works in continuous, non-terminating environments. It is an on-policy learning algorithm and interacts with the environment based on this policy. To make a decision, TD–learning evaluates the values of states, instead of state-action transitions. The algorithm takes a state as input, and the output is its state value adjusted by a factor of the difference in values between the next state and the current state. The TD–learning algorithm aims to reduce this difference, thereby reducing the error gap between the two states. The value of a state is given by,

$$V(s_t) \leftarrow V(s_t) + \alpha\,(r_t + \gamma V(s_{t+1}) - V(s_t)) \quad (3)$$

where

s_t	→ Current state
s_{t+1}	→ Next state
V()	→ State value function
α	→ Learning rate
r_t	→ Reward for action a from current state
γ	→ Discount factor

Though TD–learning does not suffer from the bias of over-optimism associated with Q–learning, it is known to sometimes get trapped in local minima while following the defined policy.

C. Double Q–learning

Double Q–learning is an off–policy algorithm like Q–learning. It uses a double estimator approach instead of the single estimator approach seen in Q–learning. This is because using a single estimator can greatly overestimate a value, while using a set of two estimators results in the samples being spread more evenly across the estimators, assuming that each of the estimators is randomly chosen at any given point in time.

The algorithm is detailed in Fig. 2, where,

s	→ Current state
a	→ Action selected for current state
α	→ Learning rate
r	→ Reward for action a from state s
γ	→ Discount factor
s’	→ Next state
a*	→ Action with max. Q–value for s’ in B
b*	→ Action with max. Q–value for s’ in A
Q()	→ Q-value function for (state,action) pair

The algorithm takes a (state, action) pair as input, and returns as output an updated Q–value for the pair. For each state, the agent randomly chooses one of the two estimators to take the action. The chosen function is updated using the estimated value of the next state from the other function. As the set of experience samples is distributed across the two estimators, each learns from a disparate set of samples. The Q–values in each function and the actions chosen by the functions are different from each other, potentially leading to underestimation of action values [14]. This enables the agent to explore much more of the state space, avoiding the overestimation bias of Q–learning. Each of the functions requires the other’s values while updating, and, hence, the estimators learn from each other.

IV. The Othello Playing Agents

For this paper, we test the performance of the three reinforcement learning agents at the game of Othello: a Q–learning agent, a TD–learning agent and a Double Q–learning agent. We also implement two fixed players against which the agents are tested: a random player (denoted by RAND) which makes stochastic moves, and a heuristic computer player (denoted by COMP) which uses a table of positional values to make moves.

```
Initialize $Q_A$, $Q_B$
  For each training episode:
      Initialize s to the starting position
      Repeat for each step:
        Choose a based on $Q_A(s,.)$ and $Q_B(s,.)$
        Execute action a
        Observe reward r and new state s'
        Randomly choose UPDATE(A) or UPDATE(B)
        If UPDATE(A) then
            $a^* \leftarrow max_a(Q_A(s',a))$
            $Q_A(s,a) \leftarrow Q_A(s,a) + \alpha\,(r + \gamma * (Q_B(s',a^*)) - Q_A(s,a))$
            Else if UPDATE(B) then
              $b^* \leftarrow max_a(Q_B(s',a))$
              $Q_B(s,a) \leftarrow Q_B(s,a) + \alpha\,(r + \gamma * (Q_A(s',b^*)) - Q_B(s,a))$
```

Fig. 2. Double Q–learning

The game board is an 8x8 checkered board. Each blank square of the board can be occupied by a black or white disc. After each move is made, the board is updated to reflect the change. At the start of each game, the board is set to the initial position of Othello. The board acts as the input for both the players.

The input state is in the form of an 8x8 vector, representing the game board. At any given point in time before the next move is made, it denotes the current state of the game. The output state is returned by the game playing agent. Once the move is made, the agent modifies the state and passes it to the board which updates itself accordingly. The following subsections describe the computer player, the random player, and the three reinforcement learning agents that play the game.

A. Computer Player

The computer player (COMP) takes an 8x8 board, represented as a 2D integer array with positions marked as black (1), white (-1) or blank (0), as input. To make its moves, COMP uses a table attributing positional values to all the squares of the game board, which has been extensively used in Othello literature [10] [13] [16] [17], as shown in Fig. 3. At each step, the computer player checks the possible moves it can make. It then checks the corresponding scores for each of the available positions using the evaluation function,

$$V = \Sigma\, C_i\, W_i$$

where,

C_i = 1 if cell i is occupied by player's disc

C_i = -1 if cell i is occupied by opponent's disc

C_i = 0 if cell i is unoccupied

W_i → Value of square i, taken from Fig. 2

V → Summation of positional values for the square

B. Random Player

The random player (RAND) chooses a cell at random to place its next disc. If the cell is occupied, or if it does not qualify as a legal move, RAND chooses another cell at random. This process continues until a legal move is made.

C. Q–learning Agent

The Q–learning agent takes the board as its input and gives as its output the cell that has the maximum Q-value among those cells on the board that are valid moves for the turn. The reward obtained, as well as the end result of the game, determine the accuracy of the Q-function, which keeps updating as it plays.

	A	B	C	D	E	F	G	H
1	100	-25	10	5	5	10	-25	100
2	-25	-25	2	2	2	2	-25	-25
3	10	2	5	1	1	5	2	10
4	5	2	1	2	2	1	2	5
5	5	2	1	2	2	1	2	5
6	10	2	5	1	1	5	2	10
7	-25	-25	2	2	2	2	-25	-25
8	100	-25	10	5	5	10	-25	100

Fig. 3. Positional values for the computer agent (COMP)

The Q–learning agent implements the Q–learning algorithm discussed in the previous section using an artificial neural network for function approximation. The agent implemented here replicates an earlier design [10] for comparison. The parameters of the neural network are a learning rate of 0.01, a discount factor of 1, and an epsilon value of 0.1 which decreases to 0 over the training period. The reward is set as 1 for a win, 0.5 for a draw and 0 for a loss. The neural network has an input layer, a hidden layer and an output layer. The input layer takes as its input the vector representing the current state of the game board. The cells with white discs are represented using a value of -1, black discs using 1, and blank cells are denoted by 0.

The output function returns a set of 64 values, with a value for each of the cells. This set is now sorted in decreasing order, and the possible moves are considered one by one until the agent arrives at a cell which is a possible move for this turn.

This is because not all actions lead to valid moves in the game. Thus the first action which leads to a valid move with highest Q-value is chosen.

D. Temporal Difference (TD) Learning Agent

TD–learning calculates its current estimate based on the present board state and previously learned estimates. It uses the value function to generate the output, which returns an afterstate value for the current state.

The inputs for the agent are the board and the set of decision parameters. The board is represented as a 2D integer array of 8 rows and 8 columns. Each cell has the value of either 1 which indicates a black disk, or -1 which indicates a white disk, or 0 which indicates a blank cell. The learning rate is set to 0.002. The reward is +1 for a win, 0 for a draw, and -1 for a loss. The TD–learning agent is an on-policy reinforcement learning algorithm, i.e., it uses a predefined policy to explore the problem state space. The policy used here is the standard greedy–ε policy.

The parameters are chosen such that they help in learning by analyzing mistakes made in the past and also by analyzing future consequences in the game. Generally, the properties of the state space are used to determine these parameters. Feature selection for a problem is primarily based on the properties defining it. For the game of Othello, the properties defining its game state are the number of white disks and the number of black disks on the board. Also, the future has to be kept in mind while making moves. Thus, we choose three parameters for the TD–learning agent: the maximum possible score that the agent can obtain on the next move, the maximum possible score that the opponent can obtain on the next move, and the ratio between the number of possible cells on which the agent makes the next move to the number of possible cells on which the opponent makes the next move. The parameters are initialized randomly to values between -1 and 1.

E. Double Q–learning Agent

Double Q–learning is an improvement on the existing Q–learning agent. A double estimator is applied to Q–learning to construct Double Q–learning. The Double Q–learning agent takes as input the board and returns a cell with the maximum Q-value. It uses two networks, A and B, both of which compute the
Q-values for all the cells for a given input and the agent chooses the action to be taken from either of those set of values in a random manner and returns the output.

The parameters of the two neural networks are set to a learning rate of 0.01, and a discount factor of 0.99 and, initially, the sigmoid activation function is used. The reward is +1 for a win, 0 for a draw and -1 for a loss. The neural networks have an input layer, a hidden layer and an output layer. The input layer takes the board state as input. For comparison against the Q–learning agent, the activation function of the hidden and the output layers is kept as the sigmoid function. We further test the performance of different activation functions for the Double Q–learning agent by also implementing the agent using the tanh and maxout functions. Finally, we add another hidden layer to the agents using sigmoid and tanh to check if this addition improves the agents' performance.

The input state for the neural network is represented using an 8x8 integer matrix to represent the current state. Thus the input represents the board at any point in time during a game before the next move is made. The cells with white discs are represented using a value of -1, black discs using +1, and blank cells are denoted by 0.

The Double Q–learning agent developed uses two neural networks for representing the two Q-functions separately. The two networks operate together. At any given step, each network generates its set of Q-values. The agent chooses from either of these in a random manner. Neural networks are preferred over lookup tables because they can easily handle an extremely large state space. The objective is to have the functions learn from separate experiences and also from each other. Any change to a function's network will eventually affect the other function's network, as Double Q–learning involves taking a network's value to update the other. Thus, the functions learn from each other. More exploration is achieved along with better learning.

V. Experimental Setup and Results

We use two fixed players to train the agents: the heuristic computer player, denoted by COMP, which uses the positional values shown in section 4, and a random player, denoted by RAND, which makes moves randomly.

The fixed players are used as opponents and as benchmark for evaluation. The Q–learning, the TD–learning, and the Double Q–learning agents are taken for training and testing. First, each learning agent is trained against another instance of itself for a set of 500,000 games. During and after this round of training, they are tested against the player RAND. Next, starting *tabula rasa*, the agents are trained for 2,000,000 games against COMP. They are similarly tested, against COMP, during and after this round of training.

The training for both rounds follow the same procedure: checkpoints are set during training, and, at every checkpoint, the learning agent is tested by playing a set of 1000 games against the designated opponent. After training, the agents are tested against the opponent for a set of 10,000 games.

Table I shows the performance of the learning agents against RAND at the specified checkpoints during the training epochs,

TABLE I PERFORMANCE OF THE LEARNING AGENTS AGAINST RAND DURING TRAINING

Training Games ($x10^5$)	Q–learning Agent			TD–learning Agent			Double Q–learning Agent		
	Won	*Lost*	*Tied*	*Won*	*Lost*	*Tied*	*Won*	*Lost*	*Tied*
0.0	33	921	46	28	923	49	21	936	43
0.1	592	376	32	537	402	61	574	389	37
0.5	769	204	27	816	156	28	803	156	41
1.0	893	56	51	927	51	22	939	47	14
5.0	941	41	18	978	12	10	986	8	6

TABLE II PERFORMANCE OF THE LEARNING AGENTS AGAINST COMP DURING TRAINING

Training Games ($x10^6$)	Q–learning Agent			TD–learning Agent			Double Q–learning Agent		
	Won	*Lost*	*Tied*	*Won*	*Lost*	*Tied*	*Won*	*Lost*	*Tied*
0.0	5	989	6	16	951	33	3	991	6
0.1	573	412	15	538	404	58	483	502	15
0.5	724	253	23	643	326	31	650	341	9
1.0	751	229	20	745	235	20	809	178	13
2.0	806	158	36	787	189	24	854	126	11

while Table II shows the performance of the learning agents against COMP at the specified checkpoints during the training epochs. The tables show the number of games won, lost and drawn by each agent against the opponent at each checkpoint during training. Table III shows the success rate of the learning agents for the 10,000-game test set after the entire training process is complete.

It can be seen that initially, when playing with little or no training, the agents lose most of the games against both opponents. Both the Q–learning and Double Q–learning agents learn quickly against RAND, owing to the fact that the random player is stochastic and does not use any strategy. The TD–learning agent does not learn quite as fast but its loss rate against the random player drops sharply after training for about 100,000 games. The agents reach a high success rate by the end of the training set of 500,000 games.

TABLE III SUCCESS RATE OF THE AGENTS AFTER TRAINING

Agent	Against RAND	Against COMP
Q–learning	0.9414	0.8059
TD–learning	0.9786	0.7874
Double Q–learning	0.9861	0.8542

The agents' performance vary significantly against COMP. COMP is a stronger player than RAND as its moves are heuristically decided. Initially, the Q–learning agent performs well, with a high drop in its loss rate but eventually slows down, with only some improvement towards the end of the 2,000,000 game set. The TD–learning agent's performance continually shows a slow and steady improvement, but finishes with the least successful performance. Like the Q–learning agent, the Double Q–learning agent also has a significant drop at the beginning and slows down after 500,000 games. But, beyond the 500,000th game, its learning outpaces that of the Q–learning agent, and finishes the 2,000,000 games set with the highest success rate of the three agents. Both Q–learning and Double Q–learning exhibit similar learning patterns, given that they are similar in structure but the difference in performance is significant, particularly against COMP.

Table IV shows the time taken by each of these agents to play a game and compares their performance based on the time taken. It can be seen that although the Double Q–learning has a much better performance than the other two, it takes about 1.46 times as long as the Q–learning agent to play each game. The TD–learning agent takes the least amount of time.

Table V depicts the success rate of the Double Q–learning agent when trained against COMP using different activation functions for a set of 2,000,000 games and tested for a set of

TABLE IV TIME TAKEN BY THE AGENTS TO PLAY A GAME (IN SECONDS)

Agent	Time taken
Q–learning	0.1487
TD–learning	0.0940
Double Q–learning	0.2168

TABLE V SUCCESS RATE OF THE DOUBLE Q–LEARNING AGENT USING DIFFERENT ACTIVATION FUNCTIONS

Activation Function	Performance against COMP
sigmoid	0.8542
tanh	0.8833
maxout	0.6724

10,000 games. Apart from sigmoid, we choose the tanh function, a widely used function, and maxout, a relatively unused function. The table shows that the agent performs best when the tanh activation function is used.

From Fig. 4, it can be seen that the performance of the agent which uses the tanh function, although off to a slower start, rises steadily and emerges with the best performance winning 89% of the evaluation games. The agent using the sigmoid function also performs well, winning 85% of the games. It reaches a high success rate early and slows down, becoming almost flat later. The agent using the maxout function is the least successful of the functions, winning 67% of the games. Its success rate keeps rising but is always behind the others.

Finally, we develop a deep double Q–learning agent to compare its performance against the Double Q–learning agent.

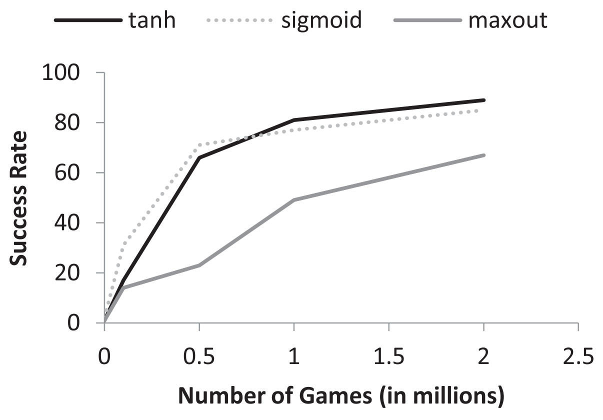

Fig. 4. Performance of the Double Q–learning when using different activation functions

The new agent uses two hidden layers. Table VI shows the performance of this agent for a set of 10,000 games against COMP after training on a set of 2,000,000 games against COMP. We implement two versions of this agent, each using a specific activation function, tanh and sigmoid, for the hidden layers and the output layer.

The Deep Double Q–learning agent performs better than its Double Q–learning agent counterpart. The agent with the tanh activation function gives the highest success rate, with the deep learning agent ahead by the end of the training set. The performance of the agents against COMP using the tanh are depicted in Fig. 5.

TABLE VI PERFORMANCE OF DEEP DOUBLE Q–LEARNING AGENT WHEN USING DIFFERENT ACTIVATION FUNCTIONS

Activation Function	Performance against COMP
sigmoid	0.8317
tanh	0.9267

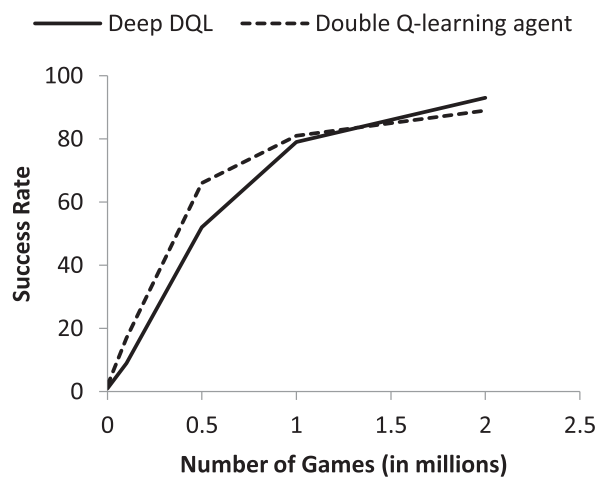

Fig. 5. Performance of the Double Q–learning and Deep Double Q–learning agents using tanh against COMP

VI. CONCLUSION

In this paper, we designed a Double Q–learning agent for the game of Othello. The aim was to develop an agent for Othello that used an off-policy reinforcement learning algorithm without suffering from the overestimation bias. We also developed other learning agents for the game and compared their performance with each other. The Q–learning and Double Q–learning agents were implemented using artificial neural networks and the TD–learning agent using decision parameters to follow a policy.

After training each agent for a set of 2,000,000 games, the Double Q–learning agent plays best against the fixed players.

However, it takes a longer time to train and make each move. The TD–learning agent plays the fastest of all the agents, but its success rate is the lowest. From the results, it can be concluded that the Double Q–learning, although slower, performs better than the two other agents, the Q–learning and the TD–learning agents. Its success rate further increases when a second hidden layer is added to the neural network and when the network uses tanh as its activation function.

Future work would delve deeper into the advantages of using the Double Q–learning algorithm and extend it to other games. The possibility of a completely unbiased off-policy reinforcement learning algorithm is still an unresolved question and could be worked upon. Further analysis on the algorithm could help in applying the extension of the standard Q–learning to the Double Q–learning algorithm to serve as a basis to extend into Multi Q–learning.

REFERENCES

[1] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, Michael Bowling, "DeepStack: expert-level artificial intelligence in heads-up no-limit poker," Science, Vol. 356, Issue 6337, 2017, pp. 508–513.

[2] Claude Shannon, "Programming a computer for playing chess," Philosophical Magazine, vol. 41, no. 7, 1950, pp. 256–275.

[3] A. Samuel, "Some studies in machine learning using the game of checkers," IBM Journal of Research and Development, vol. 3, no. 3, 1959, pp. 210–229.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller, "Playing Atari with deep reinforcement learning," in proceedings of Neural Information Processing Systems Deep Learning Workshop, 2013, pp. 1312-1320.

[5] Gerald Tesauro, "TD–Gammon, a self-teaching Backgammon program, achieves master-level play," Neural Computation, MIT Press, vol. 6, no. 2, 1994, pp. 215–219.

[6] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, 2016, pp. 484–489.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis, "Mastering the game of Go without human knowledge," Nature, vol. 550, 2017, pp. 354–259.

[8] R. S. Sutton, "Learning to predict by the methods of Temporal Differences," Machine Learning, vol. 3, no. 1, 1988, pp. 9–44.

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

[10] Michiel van der Ree and Marco Wiering, "Reinforcement learning in the game of Othello: learning against a fixed opponent and learning from self-play," in proceedings of IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, 2013, pp. 108–115.

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

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

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

[14] Hado V. Hasselt, "Double Q–learning," in proceedings of Advances in Neural Information Processing Systems, 2010, pp. 2613–2621.

[15] Hado van Hasselt, Arthur Guez and David Silver, "Deep reinforcement learning with Double Q–learning," in proceedings of the 30th Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence, 2016, pp. 2094–2100.

[16] 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, 1999, pp. 1618–1626.

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