Reinforcement Learning in Reversi

A comparative study of a Q-learning player trained with self play and minimax

Fawad Ahmad, Hamza Imran Saeed and Hafiz Areeb Asad

June 14, 2019

Abstract

The application of machine learning techniques in training agents using trial and error, without giving strategic guidance, can be used to solve sequential decision problems and can potentially find good decision making models. In this paper, Reinforcement Learning has been used to compare different strategies to train an agent to play a turn based, two player board game known as Reversi. Two different strategies were used for the comparison among players including; learning by playing against itself and also against an artificially intelligent minimax player. For the experiment, Q-learning has been used as the reinforcement learning algorithm in combination with a multi-layer feed forward neural network. The lack of time resource and computational power in training such reinforcement learning algorithms seems to be a constraint to get maximum training efficiency. After 2 million training sessions, the self-play trained agent performed better as compared to an agent trained with a minimax player, as well as random player when played against it. This shows that even without guidance of already established "intelligent" strategies to make decisions, reinforcement learning agents can learn to find strategies to solve problems.

Index Terms

Reinforcement Learning, Artificial Intelligence, Multi-agent Learning, Q-learning, Gaming, Game Playing, Othello.

I. Introduction

THERE are a number of problems which can not be solved by taking an independent decision but instead require a combination of decisions spread over time. Moreover, decisions at each step may require trade off between immediate and future rewarding results[1]. Player moves made during board games may not represent a serious decision per se, but these games are useful tools for studying decision making and developing decision and policy making systems. Finding an optimal policy which can make productive decisions by itself is an interesting area of research. In an ideal case, an optimal policy should be able to make best possible decision in every encountered state or scenario, and in the case of board games we should have the "perfect player". To find such policies for intelligent decision making, games have been extensively used

and have always been an area of interest in Artificial Intelligence. Successes such as the defeat of Garry Kasparov by IBM's Deep Blue[2] or more recently AlphaGo [3], a Go-playing AI that beat Lee Sedol, the world champion, 4 to 1 in a five-game competition, shows that using games as tools to develop and test intelligent decision making models can give valuable insights. The reason that games are useful in testing and developing new AI and Machine learning techniques is because researchers can quantify the performance of the agent with numeric scores and win-lose outcomes and compare it against humans or other AI in a finite amount of time. For this paper, the focus is on the Machine Learning technique known as Reinforcement Learning. Research in the field of Reinforcement Learning concerns itself with enabling agents to learn to make sequential decisions that maximize the overall reward [1]. Due to the sequential nature of games, they are a popular application of reinforcement learning algorithms. To learn an optimal way of playing a game, a reinforcement learning agent plays a large number of training games against an opponent. This research compares how different training opponents affect a reinforcement learning agent. In addition to this, we try to establish how the level of play (intelligence) of the training opponent affects our learning agent. This article explores the use of Reinforcement learning[1] in playing the board game Reversi. The Reinforcement learning algorithm used is Q-learning and the algorithm has previously been applied to Reversi and the Q-learning player exhibits learning behaviour [4].

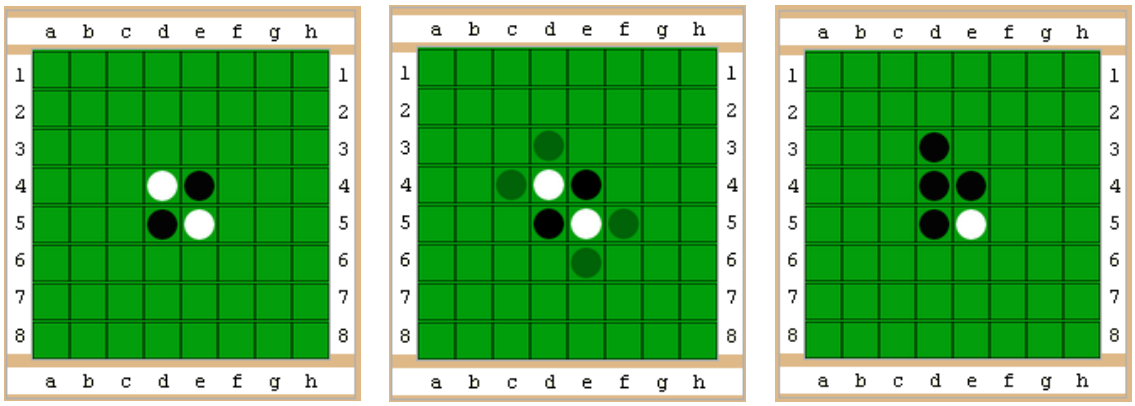
Already established AI techniques such as Minimax [5] are also used in games to have a "smart" player that can play games in an intelligent way. Using Minimax to train a Reinforcement learning agent is an interesting idea as it should theoretically give us a player that not only minimizes the opponents score but because of the exploration element of Reinforcement learning, could also improve over the traditional Minimax player. To test this, different ways to train the Reinforcement learning players and compare them are used. First, the learning agent is trained by letting it play games against itself (Self-Play). A second possible way is to train the learning agent against a minimax player, to learn from it and potentially surpass its performance. We can then play the agents against each other as well as test them against a random player to compare their performances. Thus, the article aims to have answers to the following questions

- What is the performance of the player that learns through Self-Play and assessed against random reversi player.
- How does the learning agent performance when trained by playing games against a minimax player and assessed against random reversi player?
- Is it able to beat the minimax player? What is the trade off parameters in both learning player?

In the experimental setup, an open source reversi implementation with a feed-forward neural network is used and is modified to include a Minimax player and a random player to play against. The Reinforcement learning player and the Minimax player are used in the training runs, whereas the random player is be used as a bench-marking agent.

II. Reversi

In this section we give a short description of game. The Reversi is a dual player strategy based board game. The players are assigned either white or black color. The game board is of dimension 8X8. The players take turns in placing disks, with their assigned color, on the board. Each field of the board is either empty, marked black, or marked white as shown in Fig. 1a. The players take alternate turns. In Fig.1b black selects d3 moves from available options of c4,d3,e6 and f5 shaded in grey color. After each turn, any disks of the opponents color, which are lying on a straight line, bounded by the disk just placed and another disk of the same color, are turned to the color of the current player as shown in Fig.1c d4 disk converted to black from white. If one player cannot make a valid move, (s)he must pass (skip the move). Play passes back to the other player. When neither player can move, the game ends (This occurs when all fields of the board have been marked, or when neither player can mark a field so that this would cause a non-empty field to change color). The winner is the player who has more disks that are of their assigned color[4].



(a) Initial state of board (b) Black player move (c) White player move

Fig. 1: Initial steps of Reversi Game

Reversi is a game with defined termination state. However, the state space complexity has upper limit of 10^{30}. While considering only legal moves for instance, four center squares are always non empty as shown in 1a and the possible moves are always next to existing, non-empty squares which reduces the complexity to 10^{28} in Monte Carlo analysis. If it is assumed that on average game takes 58 moves then the game tree complexity will be 10^{58} [6]. Due to this complexity, Reversi is yet to be solved mathematically [7] and there are still open question like what will be the game outcome if both player play perfect moves.

III. Minimax & Reversi

Minimax is a decision rule used in artificial intelligence and game theory that aims to minimize a measure of loss for a worst case scenario. The algorithm is a backtracking algorithm that aims to find the optimal move for a player, assuming that the opponent also makes optimal moves. The algorithm is widely used in dual player, sequential games such as Tic-Tac-Toe and Chess. Due to the sequential nature of Reversi, the Minimax algorithm can also be used to play the game and there are many open-source implementation of Minimax available for reversi as well. The algorithm has become a standard feature of computer game players. Minimax generates a game tree based on the actions available to a player at their current position, recursively goes deeper and then generates all possible opponent moves. This recursive step is then repeated until terminal nodes are reached. The terminal nodes represent finished games and scores are assigned to these nodes according to the scoring convention of the game. When these nodes are reached, the scores are propagated upwards through the game tree by assuming that each player plays optimally, i.e always choosing the action which maximizes their final score. In the case of games such as Tic-Tac-Toe (short games), the algorithm can solve the game because the number of moves required to reach the terminal nodes are very few. For games such as Reversi however, full search until the terminal nodes is not feasible because of the large number of possible moves and the tree search is instead carried out up until some fixed depth. Because the nodes at these depths are not terminal, they do not have a score associated with them as the game is not finished and, a method for scoring intermediate states is required. For board games like checkers, chess, and Reversi, the number of intermediate states is immense, one for each unique board state, and a model for evaluating board positions is required [8]. There heuristic function which is used to determine the state of the board and make a decision for the algorithm can take several factors and features of the game into account. The heuristic function used for the algorithm, which is an implementation of the Minimax with additional Alpha-Beta Pruning, is based on immediate mobility (number of moves available at each turn). There are other well researched and effective heuristic functions for Minimax as well, but for the purpose of this experiment, given that those heuristic functions can add to the computational complexity, the above mentioned heuristic is used.

IV. Reinforcement Learning

Reinforcement Learning is a machine learning technique that enables an agent to learn by trial and error using feedback from its own actions and experiences. The player is a decision making agent that decides on an action to take and receives a reward for its actions from the environment. The aim is to learn the best policy after trial and error runs, i.e which set of actions results in maximum rewards [9]. The process of Reinforcement learning can be illustrated in Fig.2. Reinforcement learning differs from the supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent

Fig. 2: Reinforcement Learning

decides what to do to perform the given task. In the absence of training dataset, the agent learns from experience and it can go on to give better results than instances where the agent is trained on already established data (best moves in this case).

V. Reinforcement Learning & Reversi

In this section, we discuss reinforcement learning and how it is applicable to Reversi. In reinforcement learning, an agent is the learning system which makes decisions in the form of actions or moves in case of reversi. Based on performed action the environment which is a 8X8 board in this case provides feedback. The feedback environment provides can be a reward or a penalty. After trial and error agent learns which set of sequence results in maximum rewards.

The goal is to find an optimal policy which agent can follow to relate states to actions. An optimal policy defines an actions to take in any given state. The expected rewards will be cumulative over time if agent follows optimal policy. Such sate action pair is called Q-values denoted by {s,a}.

$$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^{i} r_i | s_0 = s, a_0 = a, \pi] \tag{1}$$

Where E is expectancy operator, γ^i is discount factor, r_i is reward, π is policy while and s_0 and a_0 are initial state and action respectively.

A. Reinforcement Learning Algorithm

When an agent plays against another player, the result of the move taken by the agent cannot determine the result because the agent has to wait for opponent players move. Moreover, the direct relation between state and action can not be established as the same state and action environment might give different result because there is an other entity involved, i.e an opponent player that also influences the game state. The algorithm that relates state and actions is called Q-learning

algorithm [10]. In this algorithm agent estimates values action values against given state and selects the best action (max).

$$\pi(s) = \arg\max_{a} \hat{Q}(s, a) \tag{2}$$

However, this will always be an estimated optimal policy π and agent will never be able to learn new better strategies. To overcome this problem ϵ-greedy exploration is introduced which performs random action with ϵ probability instead of following the learned policy. The probability of ϵ is decreased over time during training.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[r_t + \gamma \max_{a} \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t)] \tag{3}$$

while α is learning rate that controls the difference between current Q-value and accumulated Q-values estimates. This is called temporal difference algorithm [11]. While γ^i control how much policy foresees future rewards instead of immediate rewards and s_{t+1} is next state.

The game problems which are less complex can be solved by storing all possible state action pair in a lockup tables. However, the Reversi game tree complexity is around 10^{58}. Which causes two main problems First, the state space is vary large. Second, The agent will lose it's ability of generalization because agent might be presented with a scenario for which it is not trained during training session. Due to these reasons the neural networks are used to estimate states to corresponding Q-values and generalization property of neural networks helps in overcoming unseen states. In learning phase, neural networks learns the mapping between states and values by presenting target to neural network using (2). Also the α learning rate is set to 1 to because in Multi Layer Perception Feed Forward Neural Network [12] there already exists learning rate known as η which simplifies equation (2) to

$$\hat{Q}(s_t, a_t)^{new} \leftarrow r_t + \gamma \max_{a} \hat{Q}(s_{t+1}, a) \tag{4}$$

The Q-learning is implemented in NN by having inputs as states and outputs as actions.

B. Application to Reversi

The Reversi is a turned based, two player game and implementing the Q-learning algorithm on the game requires the opponent player's move before learning board states. Therefore, the values of the state action pair are learned at start of our agents turn. In order to execute action, the agent performs the following steps:

1) Assess current s_t
2) Use NN to compute $\hat{Q}(s_t, a_t^{'})$ next allowed actions $a_t^{'}$ in current s_t
3) Choose action a_t following policy π
4) Compute target value of previous state action pair $\hat{Q}^{new}(s_{t-1}, a_{t-1})$ by (2)
5) Use NN to calculate current approximate value of previous state action pair $\hat{Q}(s_t, a_t^{'})$.

6) Adjust network weights by back propagating error $\hat{Q}^{new}(s_{t-1}, a_{t-1}) - \hat{Q}(s_{t-1}, a_{t-1})$
7) $s_{t-1} \leftarrow s_t, a_{t-1} \leftarrow a_t$
8) Execute the action a_t

C. Learning from Self-Play

In Self-Play, both agents are reinforcement learning agents. They are played against each other and each move is selected based on current best valid move. During learning both agents follow the best policy 90% of time and make 10% random moves.

D. Learning from Minimax player

In Minimax Learning, the agent is trained by playing against a Minimax player with depth in the range of 4 and 6 levels deep. Although alpha beta pruning [13] is used, increasing the depth of the Minimax player can take large amount of time during the training session because Minimax has to populate the game tree and because reversi can have many possible moves at each depth, thus the game tree increases exponentially. This means that having a depth larger than 6 is not computationally feasible on conventional computer thus the depth was limited to this number.

E. Random player

In order to benchmark the performance of both agents, a random player plays against other agents by selecting a random move from a list of available valid moves. This agent is used to make an unbiased comparative analysis because the trained agent should be played against a player that it had not seen before so that any form of bias(as a result of training against the Minimax player) could be removed. Similarly, agents trained using self-play were played against the random player as well so that it could be analysed without bringing in any form of bias. Thus, there is no bias when a random agent is used, and the random agent provides a new challenge for the reinforcement players, one that they did not train against. Now the performance of agent can be estimated by playing games, selecting a number of samples and averaging the results.

VI. Experimental and Results

In first experiment the Self-Learning agent was trained for 70,000 epochs and during training it was tested by playing against random player. Due to large number of oscillation and data points the start and end of this training session is plotted in Fig 3 and Fig 4.

Fig. 3: Self-learning vs random player for first 5,000 epochs

In Fig 3, performance of Self-Learning agent is shown for first 5000 epochs. In this plot the number of games won oscillates between 25 and 45, with an average around 40 and after 4000 epochs starts shows a slight improvement. In this plot, the testing, i.e by playing against the random player, is done after each epoch.

Fig. 4: Self-learning vs random player for last 20,000 epochs

In Fig 4 ,agent continues the same trend until 60,000 epochs. In this plot, the testing is done with a 100 epoch interval.

In second experiment, the minimax agent was trained for 5,000 epochs. As minimax training is computationally intensive and during training it was tested by playing against random player.

Fig. 5: Minimax vs random player for 5,000 epochs

In Fig 5 performance of minimax trained player can be seen. It is similar to the result we got for the self-play trained agent, with slight improvement after 3500 epochs. In third experiment after

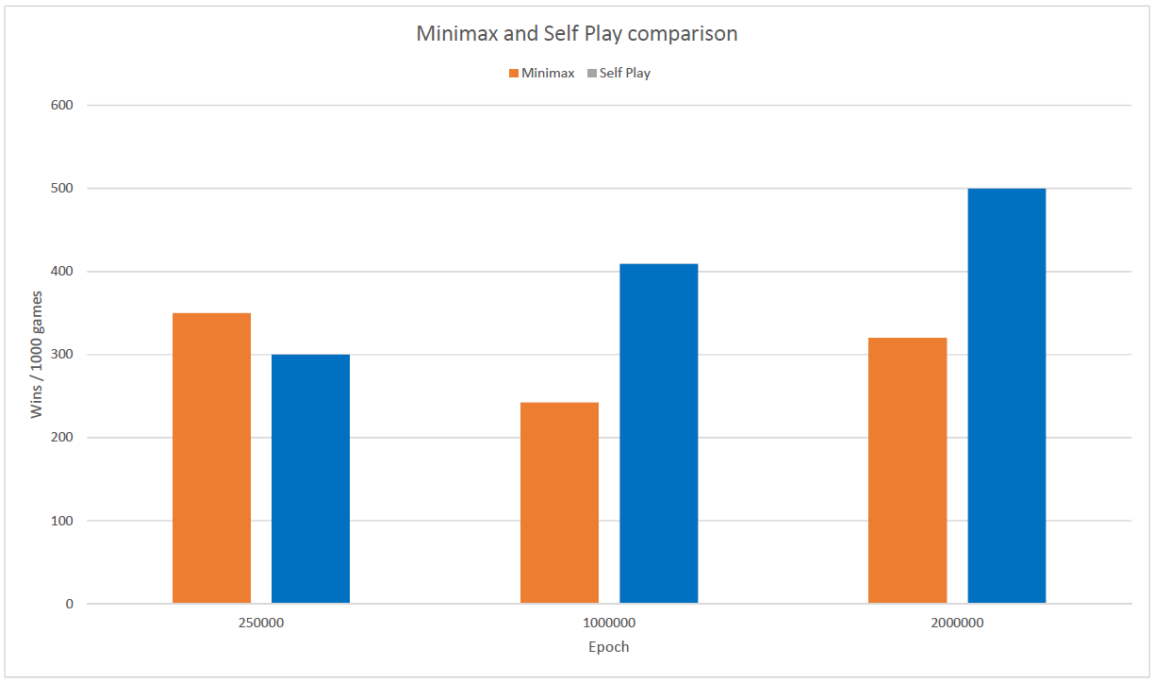


Fig. 6: Self-learning and Minimax agent against Random player for 1000 games during training

70,000 epochs, the minimax agent played against the Self-Learning agent for 100 games and it was found that Minimax agent was not able to win a single game. Moreover, it was observed that Minimax agent lost due to weak heuristic function based on maximum score.

Based on previous experiments and results, the minimax agent heuristic function was improved based on mobility that limits the opponents moves.

In the fourth experiment shown in Fig 6, the modified minimax agent and self play agent was again trained for 2 Million epochs from beginning. To measure the performance, both the agents played 1000 games against the random player at three points during the training. A histogram is made to show the performance results of both agents against random player in Fig 6.

VII. Discussion and Reflection

Usually when Q-learning is used, the estimated Q-values are stored somewhere during the estimation process and thereafter. A simple way to store these Q-values is a lookup table with a separate entries for state-action pairs. The problem with this method, in the case of Reversi is the space complexity. When the state-action space is large, the learning can be very slow and very large tables are required to store the Q-values, and in the worst case these tables may be too big to be stored in a computer memory. This is the case with Reversi, where the number of entries is 10^{58} [6] and is not feasible to store in memory.

To deal with this problem, a function approximation method such as a Neural Network can be used and that is what we use in this experiment. During the run time of the Q learning algorithm, the NN learns a mapping from state descriptions onto Q-values. This is done by presenting target Q-value to neural network using (2) and using the backpropagation algorithm to minimize the error between the target Q-value and the estimated Q-value computed by the NN. In general, a problem with large state-action spaces is that a NN is trained by visiting a small part of the state action space. However, the NN can generalize and based on previous experience with visited state-action pairs, the NN is able to give an estimate of the Q-value for an an unknown state-action pair. In our assessment, the NN is able to generalize because as the training progresses, the internal layers of the NN learn to extract features that are useful in assessing the Q-values of state-action pairs. Based on these self-discovered features the final layer of the neural network will make estimates of Q-values [4].

Using NN to store the Q-values can solve larger problems as compared to Q-learning using a table, but it is not guaranteed in the case of a NN, the Q-values are not guaranteed to converge. The problem is that the NN performs non-local changes to the Q-function, while Q-learning requires that updates to the Q-function are local, thus when the value of a state-action pair is updated, the learned values of some other state-action pairs may be destroyed and this is why the NN method is not guaranteed to converge to the actual Q-values.

There are two main ways that the Feed Forward NN could have been made for the experiment. One is to have a single Neural Network where there is a distinct output for each action. The network has 64 input units (corresponding to the 64 board squares) which represent the state of the environment. The activation of an input unit is 1 for a players own disc, 1 for a disc of the opponent, and 0 for an empty square. The network has one hidden layer of 44 tanh (hyperbolic tangent) units and an output layer of 64 tanh units which correspond to an action. The value of an output unit is between 1 and 1 and corresponds with the Q-value of a move. The learning rate for the NN is set to 0.1 and no momentum is used. The weights of the network are initialized to random values drawn from a uniform distribution between 0.1 and 0.1. This is the method we use in our experiment. The other method is to use a distinct Feed Forward NN for each action and in this case the number of neural networks is equal to 64. Each network has 64 input units. In the same way as for a single-NN Q-learner these units are used to present the squares of the board to the player. Each network has a hidden layer of tanh units and an output layer of 1 tanh unit. The rest of the properties are the same as described above and thus the only difference is how the values are stored [4].

As far as the action selection is concerned, there is a trade-off between exploration and exploitation that has to be made as the agent wants to exploit what it has already learned and get a positive reward, but exploration can lead to better action selection in the fututre. There are many ways and established methods to balance between exploration and exploitation in agents and we use the softmax action selection method, where the agents choose actions in a probabilistic way, based on their estimated Q-values using a Boltzmann distribution [4]. The initial value for exploration as a result of this is high but it is eventually decreased over time and the gradual transition is from exploration to exploitation.

Furthermore, in this implementation, players reward is 0 until the end of the game as this makes sense according to the the rules of the game. When a game is completed, the agent is given a reward of 1 if it wins, 0 if it draws and 1 if it loses. In this way, the agent aims to choose optimal actions leading to maximal reward. The values the parameters for the Q-learning algorithm may not be optimal as they have been chosen experimentally. The learning rate of the algorithm is set to 0.1 and does not change during the learning process. Lower rates made the training learning very slow and time was a constraint. The discount factor is set to 1 and this because we only care about winning and not about winning as fast as possible.

VIII. Conclusion and Future Recommendations

In first two experiments, the agents were tested against a random player which is not a best player as it's moves are random. However, the Minimax performed better against the random player in same number of epochs (5000 epochs). Surprisingly, in third experiment where Minimax

agent played against self play agent, the self play agent always won the game and Minimax agent was not able to win a single game. This behavior can seem strange at first but when the working of both learning system were observed it was clear that Minimax agent always strives for maximum score while Self-learning agent strives for a different strategy, that is striving to capture corners before terminal state of game is reached. It is now clear that when Minimax agent plays against the self play agent, the self play agent quickly captures the corner points and ends the game which is a good strategy learnt by the self-learning agent. It can be concluded that, performance of Self play agent is better when it plays against the random player in-comparison with the Minimax agent. Also, the Self-Learning agent performs best against Minimax agent due to tricky move of corner capturing. In order to investigate this hypotheses, in fourth experiment, the Minimax agent was modified with improved heuristic function that limits the movement of opponent and resist in capturing corners, this significantly improved the performance of Minimax agent but still was not able to surpass Self-Learning agent performance.

There are various ways that the agent could have been trained which were not explored in this paper but can give better results. For example, the depth of the Minimax algorithm can be increased which might will give a better minimax training agent. Furthermore the MLP neural networks can be replaced with deep neural network for deep Q learning which has had success in playing games such as Go. Finally there exists detailed opening books that have huge library of moves and known best moves at each position which can be utilized for initial training in conjunction with other machine learning techniques to achieve a favourable trade-off between intelligent decision making and training time.

REFERENCES

[1] R. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, vol. 9, p. 1054, 02 1998.

[2] S. Hamilton and L. Garber, "Deep blue's hardware-software synergy," *Computer*, vol. 30, no. 10, pp. 29–35, 1997.

[3] F.-Y. Wang, J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, and L. Yang, "Where does alphago go: From church-turing thesis to alphago thesis and beyond," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 2, pp. 113–120, 2016.

[4] N. J. 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.

[5] M. Willem, *Minimax theorems*. Springer Science & Business Media, 1997, vol. 24.

[6] L. V. Allis *et al.*, *Searching for solutions in games and artificial intelligence*. Rijksuniversiteit Limburg, 1994.

[7] D. W. Russell, *The BOXES Methodology: Black Box Dynamic Control*. Springer Science & Business Media, 2012.

[8] K. T. Engel, “Learning a reversi board evaluator with minimax.”
[9] O. Gällmo, “Lecture notes in natural computation methods for machine learning,” February 2019.
[10] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
[11] R. Sutton, “Learning to predict by the method of temporal differences,” *Machine Learning*, vol. 3, pp. 9–44, 08 1988.
[12] D. Svozil, V. Kvasnicka, and J. Pospichal, “Introduction to multi-layer feed-forward neural networks,” *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.
[13] S. H. Fuller, J. G. Gaschnig, J. Gillogly *et al.*, *Analysis of the alpha-beta pruning algorithm*. Department of Computer Science, Carnegie-Mellon University, 1973.