university of groningen / faculty of science and engineering

Reinforcement Learning in the game Lines of Action: Input Representations and Look-ahead

Bachelor's Project Thesis

Remo Sasso, s2965917, r.sasso@student.rug.nl
Quintin van Lohuizen, s2701782, q.y.van.lohuizen@student.rug.nl
Supervisor: Dr M.A. Wiering

Abstract: This paper investigates the application of reinforcement learning to the game Lines of Action by using a multi-layer perceptron (MLP). It investigates which training and testing method leads the agent to perform best, while applying a temporal difference learning algorithm. Three opponents are created in order to determine the most suitable training opponent: a random opponent, a fixed opponent and the agent itself. Additionally, different kinds of input representations are fed into the multi-layer perceptron, to see whether they affect the performance if used during training. Lastly, a testing method is introduced where the agent uses a look-ahead strategy. This allows the agent to perform a deep search, which may affect its performance. For this research the temporal difference learning method TD(0) was used in combination with an MLP. By using the training opponents as testing opponents, the resulting performances showed that the agent learns best against itself. The look-ahead play was tested in an identical manner, which produced a significant improvement in performance against opponents that are not completely random. Finally, we found that by training the agent with different game state representations, performance significantly increases when trained against a fixed opponent or the random opponent.

1 Introduction

The field of reinforcement learning (RL) is concerned with digital agents living in a sequential time frame (Sutton and Barto, 1998). These agents are put into an environment of which they may have no prior knowledge whatsoever. By taking the actions made available to them, they explore this environment and learn how to function in it. Taking such an action leads to a reward or punishment, which teaches the agent how beneficial it was to take that action. The goal of these agents is to maximize the rewards they obtain in the future, which in turn, optimizes their behavior for the specified problem. In that situation a very popular testing environment in RL is games. As games either already are, or can easily be broken down into a sequential time frame, they serve as a perfectly suitable RL testing environment.

Due to their discrete and simple origin, board games were a beloved application in the early years of RL. Samuel (1959) was one of the first ones to succeed in creating a program capable of learning to play a board game, checkers in his case. This publication was important, as it introduced methods later coined as temporal difference learning methods (TD-learning) by Sutton (1988). These methods turned out to be very beneficial for agents learning to play a board game, as it was the core component of the remarkable world-champion Backgammon program by Tesauro (1995). Tesauro showed that by combining a multi-layer perceptron with an RL method, above human level performance can be achieved, allowing agents to compete with experts in the game. This also applies to other board games, such as chess Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, Lillicrap, Simonyan, and Hassabis

(2017b), the game of Othello Buro (2003), checkers Schaeffer, Hlynka, and Jussila (2001) and Go Silver, Huang, Maddison, Guez, Sifre, van den Driessche, Schrittwieser, Antonoglou, Panneershelvam, Lnactot, Dieleman, Grewe, Nham, Kalchbrenner, Sutskever, Lillicrap, Leach, Kavukcuoglu, Graepel, and Hassabis (2017a). Despite many successful attempts on the mentioned games, there are still games out there which have received very few, or no attempts at all. Winands, Kocsis, Uiterwijk, and van den Herik. (2002) were the only ones who successfully applied TD-learning on a less well-known game called Lines of Action (LOA). They showed that using TD-learning for tuning the weights of their approximator improved their agent's performance significantly.

In this research we will teach an agent to be able to play LOA with model-free RL, by making use of TD-learning. That is, the agent will have no prior knowledge and will learn over time how to play. In order for an agent to learn to play a game with RL, it is required to play a massive number of games against a training opponent. We will compare the agent's performances of training against several opponents. Additionally, we will investigate whether enabling the agent to look several steps ahead affects its performance. The look-ahead function took inspiration from the TD-Leaf algorithm introduced by Baxter, Tridgell, and Weaver (1998). Finally, we will experiment with what we like to call meta-features (i.e. additional information about the state of the game), in order to examine whether and which meta-features affect the performance.

In order for the agent to learn to play LOA, an opponent must be found that teaches the agent to perform best. There are several possibilities for this, which we can categorize into: random opponents, fixed opponents and self-play. A random opponent merely picks random moves, which is a quick way for the learning agent to learn how to win. Fixed opponents are more challenging opponents which use a heuristic relevant to the game. Hence, the agent will lose more often than with a random opponent and thus also learns more frequently from losing. Lastly, the agent can learn from playing against itself, which gives the agent an opponent which is always equally as good as the agent itself. For attempting to improve the agent's performance even further, we can supply the agent with the earlier mentioned meta-features and look-ahead function. These approaches and the differences among them, will be the focus of this paper. This will help us answer the following research questions:

- How do the performances of the learning agent trained against different opponents compare to each other?
- How do the performances of the learning agent compare, when playing with and without a look-ahead function?
- How do the performances of the learning agent compare, when training with and without additional meta features of the game?

Previous research has shown that with TD-learning, in the game of Othello by van der Ree and Wiering (2013), self-play results in the best performing agent. We are interested in whether this also applies to LOA. Winands et al. (2002) decided to include meta-features in their valuation function, thus contributing to estimations. In this paper, similar features, are used as additional information for the neural network. The input layer has extra neurons for this information, as well as the output layer. Therefore this extra information also contributes to the valuation of the game state. The output layer will predict the value of the extra input information. As we will include the features in back-propagation, it will be interesting to observe how and whether performance is affected by this approach. TD-Leaf applied during training, has shown to be improving performance significantly by Baxter et al. (1998), therefore it will also be compelling to see whether the tree-search method for finding the best move improves performance when exclusively used in testing.

The remainder of this paper is organized as follows. In section 2 we explain LOA. In section 3, the theory behind the used methods is described. Section 4 describes the experiments that were performed and section 5 shows the results. A discussion will be presented in section 6, after which the conclusions will be stated in section 7.

2 Lines of Action

Lines of Action is a zero-sum board game, played on an eight by eight board. The information for

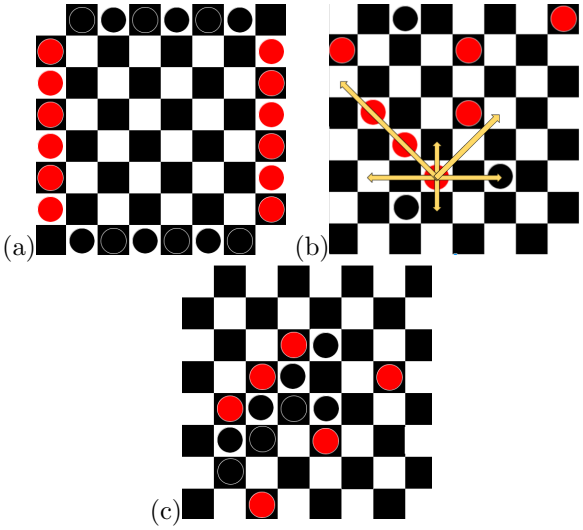


Figure 2.1: A screen shot of: (a) Beginning state of the game, each agent has twelve stones at its disposal. (b) Middle stage of the game with the possible moves of a single marked stone. (c) Terminal state of the game where black got an 8-connectivity cluster and wins.

both players is complete, as there is no hidden element in the game. It is played by two players alternating turns in which one stone is moved. Figure (2.1a) shows the beginning state of the game. A stone may only move in a straight line. The number of steps a stone may traverse, is equal to the total number of stones in that direction (no more and no less). Figure (2.1b) shows all possible moves for the stone that is marked. This includes jumping over a friendly stone, whereas you are not allowed to jump over enemy stones. An enemy stone is captured when a friendly stone lands on top of it. The game is won by the first agent to connect all of its stones or the agent who has one stone left. This is validated with 8-connectivity, therefore the allowed connections are: horizontal, vertical and diagonal. Simultaneous connection of clusters is considered a draw. Figure (2.1c) illustrates a cluster (terminal state) formed by the black player. If both agents alternate with a single stone between two locations three times, the game is also considered a draw, since no progress is made. The branching factor is the amount of possible moves per board position, Winands (2000), found it to be 36 in the beginning of the game and 30 on average.

3 Methods

In this section the methods used in this research are described with the corresponding theory. We will first introduce reinforcement learning and the sequential decision making process that comes along with it. What follows is a description and discussion of temporal difference learning and the approximator used in this research. Next, the application of these methods to LOA is presented, and lastly we discuss in more detail the input representations and the look-ahead strategy.

3.1 Reinforcement Learning

Reinforcement learning generally applies to sequential decision making problems. In such problems there is an agent situated in some environment and this agent has to take a decision every time step. These decisions are considered as actions, which lead to new states and a reward or punishment. In a trial-and-error fashion, the agent attempts to solve the given problem, while being guided by the rewards and punishments. As such, over time the agent will learn what the optimal policy is (i.e. most desired sequence of actions) in order to solve the problem, or rather, in order to maximize the total reward it obtains in future time steps.
Such sequential decision making problems are usually converted into a mathematical construct called a Markov decision process. These are constructed as follows. Take S to be a finite set of states $s \in S$; Take A to be a finite set of actions $a \in A$; Now take $P_a(s, s') = P_{a_t}(s_t, s_{t+1})$ to be the probability of taking action a in state s at time t to result in state s' at time $t + 1$; Next we take $R_a(s, s')$ to be the reward corresponding to the transition from state s into state s' after taking action a: Finally, we take $\gamma \in [0, 1]$ to be the discount factor, which specifies the degree of how future rewards are considered less important than direct rewards. What remains is the policy π, which tells the agent what action a to take in a certain state s at time t: $a_t = \pi(s_t)$.
Now the goal of the agent is to learn the optimal policy: the policy with the maximum cumulative reward. Therefore, for each policy π starting in state $s_{t=0}$ the expected cumulative reward $V^\pi(s)$ has to be determined. This is defined as follows:

$$V^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\Big] \quad (3.1)$$

In this equation, an action a in state s at time t is determined by $\pi(s)$. $E[...]$ is defined as the expectancy operator. The optimal policy for the starting state is the one where the expected cumulative reward is the greatest.

3.2 Temporal Difference Learning

Temporal Difference Learning is a class of RL methods which train an agent to predict a quantity that depends on future values. The agent thus learns to predict the valuation of a given state, based on the valuation of following states. The difference between the valuation of a given state and its following state(s) is called the temporal difference error (TD-error). The valuation of a state at a given time step is updated with respect to the state of the next time step, in order to decrease the TD-error between them. In this manner, the agent learns to predict the total amount of reward expected over the future.

The simplest method of TD-learning is called TD(0) (Sutton, 1988). In this method, the valuation of a given state s_t is updated with respect to *only* the valuation of state s_{t+1} and the reward r_{t+1}, hence only the first following state. In other TD-learning methods, future states (s_{t+n}, where $n > 1$) are also taken into account such as in TD(λ) (Sutton, 1988) or TD-Leaf (Baxter et al., 1998). Also, in contrast to other TD-Learning methods such as Q-learning (Watkins and Dayan, 1992) where the valuation of a state-action pair is predicted, in TD(0) we only predict the valuation of the state. These are the main reasons TD(0) is considered the simplest method of TD-learning.

Before we discuss the valuation function of TD(0) any further, it is important to note that when an agent is playing a two-player game, the state s_{t+1} denotes the state *after* the opponent also executed an action. What follows is that the predictions made are non-deterministic, as the transition to the resulting state might have a positive or negative reward depending on the opponent's action. This, however, poses no further problem for the TD(0) algorithm as such state transitions will eventually acquire lower valuations in comparison to state transitions which have yielded positive rewards more consistently.

The TD(0) algorithm is defined as follows:

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (3.2)$$

Where $V(s_t)$ is the valuation of state s at time t; γ the discount factor described in section (3.1); α is denoted as the learning rate, where $\alpha \in [0, 1]$; And finally, r_{t+1} is the reward resulting from the state transition, which therefore is the reward acquired in time $t + 1$. However, when an agent uses this algorithm for learning, it also has to determine which action to take in a given state. One would assume that a good policy would be for the agent to take the action that results in the state with the highest state valuation. However, with such a policy the agent will follow one specific strategy which only takes actions according to the states with the optimal estimated valuation. This means that the agent will not be able to learn any other strategies and will therefore not be able to find a better one. This is a well known problem in RL called the Exploration vs. Exploitation dilemma: finding the right balance between exploring new and exploiting already known strategies. One solution for this problem is a policy called the ϵ-greedy policy. This policy makes the agent act greedy, in taking actions that have the highest valuation estimates. However, with a probability of ϵ, the agent takes a completely arbitrary action, in order for it to explore different strategies. Here ϵ is set to a value in the range $[0, 1]$ and may change over time.

3.3 Function Approximator

Some problems allow the values of each state transition to be stored in a look-up table. However, this is simply not possible with larger space complexity problems, since storing such a huge quantity of data is not feasible. Lines of Action has a space complexity of 10^{23} (Winands, Uiterwijk, and van den Herik, 2001), therefore storing all states and their valuations is not possible in any case. Additionally the agent might encounter input patterns on which it has not trained, hence it would have no knowledge of which action is best to take in these states. Therefore a generalization over the input patterns is desirable. This can be achieved with an approximation of the valuation of the input patterns. An

approximation ensures that the agent can deal with states that it has not seen before. A multi-layer perceptron (MLP) can estimate the valuation of an input pattern instead of using a look-up table. This enables the agent to generate sensible generalizations over the input patterns, when it is trained sufficiently. The neural network learns to map the state descriptions to the valuation of those descriptions. These descriptions can be the plain game state (8×8 board) or include extra descriptors, such as the amount of moves a player is allowed to make. A target value (TD-target) is computed according to the TD-learning algorithm described in 3.2. The learning rate 'α' in Equation 3.2 is set to 1.0, as the neural network already includes a learning rate. Therefore we can simplify Equation 3.2 to Equation 3.3.

$$V^{NEW}(s_t) = r_{t+1} + \gamma V(s_{t+1}) \tag{3.3}$$

Batch training is used to decrease the time needed for training. A large number of games is played, storing the game state before and after each move. The neural network valuates the first state of the state-pair and determines the valuation of the next state. The valuation of the next state and its reward form the TD-target $V^{NEW}(s_t)$ of state s at time t. The TD-error between the TD-target and the valuation of the first state is determined and back-propagated through the neural network. This changes the valuation of the first state in the state-pair accordingly. Figure 3.1 illustrates the structure of the network used in this research.

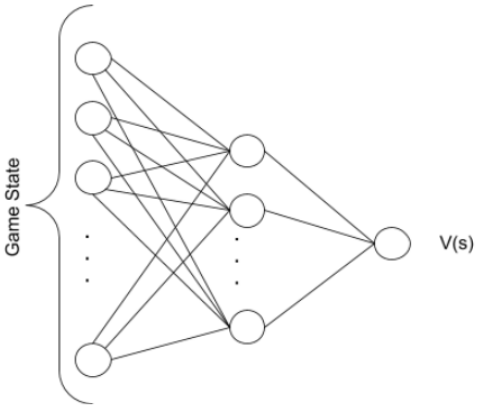


Figure 3.1: The input layer is the size of the representation of the game state. The state is represented as an array of length 64, in which a friendly stone has a value 1, an enemy stone -1 and a blank square 0. The hidden layer contains 50 neurons and the output layer has one neuron, which outputs the valuation of the game state.

3.4 Application to Lines of Action

As mentioned earlier, an important fact to consider in LOA is that it is a two-player game: only after the opponent has made a move, the agent can find itself in a new state. In a given state, we want to determine what action to take, based on the valuations of the states all actions lead to. Therefore we need to introduce afterstates: the state of the game after the player has made a move, but before the opponent makes a move. As such the best move in a state is the afterstate with the highest valuation. Rewards are only provided in a terminal state (win or a loss) and, as these states have no next state, valuations of terminal states equal the reward's value. In all other cases the reward is zero, which is why we can simplify Equation 3.3 even further:

$$V^{NEW}(s_t) = \begin{cases} r_{t+1}, & \text{if } s_{t+1} \text{ is terminal} \\ \gamma V(s_{t+1}), & \text{otherwise} \end{cases} \tag{3.4}$$

Here the reward r_{t+1} is determined by:

$$r_t = \begin{cases} 1, & \text{if win} \\ 0, & \text{if draw} \\ -1, & \text{if loss} \end{cases} \tag{3.5}$$

Every turn excluding the first, the agent performs:

1. Observe the current state of the game s_t
2. Determine for each afterstate s'_t obtained from s_t, the valuation $V(s'_t)$
3. Perform action a according to a policy π
4. Compute the target value of the previous afterstate $V^{NEW}(s_{t-1})$, using formula (3.4)
5. Determine current valuation of the previous afterstate $V(s_{t-1})$
6. Determine the error
   $error \leftarrow V^{NEW}(s_{t-1})$ - $V(s^{t-1})$
7. Use the error to adjust the neural network via back-propagation

8. $s_{t-1} \leftarrow s_t$

9. Execute the action resulting in the desired afterstate s_t

3.5 Learning from Opponents and Self-Play

In order to find the most suitable opponent for the agent, we compare three opponents it can train against when playing training games. The following three opponents are used in this research: a random opponent, a fixed opponent using a heuristic, and the agent itself.

1) *Learning from a random opponent:* A completely arbitrary opponent will allow the agent to learn how to win relatively quick, as there is no real competition from the opponent.

2) *Learning from a fixed opponent:* A fixed opponent is more challenging than a random opponent as it may use a heuristic that works well for the game. As such, the agent will learn how to deal with an opponent that is more likely to win. Therefore, the agent will also learn from losing, which in turn allows for better strategic generalization possibilities.

3) *Learning from Self-Play:* When learning from self-play, both agents share the same neural network and both follow the algorithm described in section 3.4. This allows for a doubled amount of training material from a single game, as well as providing an opponent that is always equally as skilled as the agent.

3.6 Look-ahead

Baxter, Tridgell, and Weaver published a chess playing program in 1997 where they combined TD-learning with minimax search, after which they introduced the TD-Leaf algorithm (Baxter et al., 1998). This algorithm integrated the original TD(λ) algorithm with game-tree search. By enabling an agent to perform searches in the game, they gain an important ability we generally also make use of in board games. Claims have been made that this is one of the primary reasons for programs to be able to compete with the best human players in the world, as has been shown in games as Backgammon (Tesauro, 1995), chess (Silver et al., 2017b) and Go (Silver et al., 2017a). In this algorithm they define the TD-target to be the best move discovered by a limited ply game-tree search, instead of the difference between the valuation of two sequential states. The best move that has been discovered in the specified depth is therefore one of the search-trees leaves, hence the name TD-Leaf. The valuation of this leaf is then used to update the valuation of the current state.
This inspired us to take this game-tree search idea for finding the best move (or leaf) in the specified depth, in order to enable our agent to perform deep searches. However, due to the exponential increase in the required computational power of this algorithm, we decided to take the core concept of this algorithm and convert it into a suitable tree search for the agent, merely used for testing games. In this paper we used a 3-ply search with pruning included, working as follows:

1. Observe the current state of the game and determine all possible moves (afterstates), of which three afterstates are chosen with the respective highest valuations.

2. For each of the three afterstates resulting from step 1, determine the afterstate of the opponent with the lowest valuation (highest for the opponent).

3. For each of the opponent's afterstates determined in step 2, generate all possible afterstates and determine which move has the highest valuation of all the generated afterstates in this step. If there are multiple leaves with this valuation, pick one arbitrarily.

4. The best afterstate determined in step 3 resulted from one of the afterstates determined in step 1. Trace back to that corresponding root-move and execute that move.

A simplified illustration of this tree search can be found in the appendix (Figure A.1). In order to make this search deeper, one could repeat steps 2 and 3. One thing worth emphasizing is that in this search, it is assumed that the opponent takes the best possible move which presumably will cause issues if the opponent for example uses a random strategy.

3.7 Input Representations

The performance of the agent is determined by the information it receives, before we mentioned that the only information it receives is an array with a length of 64, representing the state of the game. The performance might increase when the neural network is provided with more information about the game state. The extra information considered in this paper is:

1. Immobility
2. Degree of Freedom
3. Concentration
4. Biggest Cluster

Figure 3.2 shows how, the additional neurons for capturing the additional game information, are added to the neural network. Information is fed as input patterns to the input layer and predicted in the output layer. Caruana (1997) has shown that adding additional information to the output layer without adding it to the input layer of the network can increase performance. For some problems the additional information is not available during run time and therefore ignored. Caruana (1997) showed that for these problems a higher performance can be achieved by simple adding this information to output layer. But In our case, the information is available in run time and can be stored. Therefore we will add the extra neurons for this information, in the input- and output layer. Let state s_t be a state in where the agent has to take an action. And let the *previous best afterstate*, be the state achieved by performing the best action according to its policy in s_t. State s_{t+1} is reached when the opponent has made a move. Now the agent has to choose a new action. Let the best action in s_{t+1} be the *afterstate*. The *previous best afterstate* and *afterstate* form a pair, which will be used in batch training. The extra input information and valuation from the afterstate are taken to be the target value of the previous best afterstate. Therefore the error between the two states can be determined and used for backpropagation. The goal is to predict the values of the extra inputs so that the backpropagation algorithm can change the weights of the neural network accordingly. All input features described in this section are normalized between a value of 0 and 1, to make sure the input values do not have to much impact on the network.

Immobility: A common practise in LOA is putting up walls for the other player. This ensures that the other player has stones, that are completely immobile. Two extra input and output neurons are added. The first in the input layer is the amount of the immobile stones belonging to the learning agent, while the second neuron is that of the opponent. The output layer consists of three neurons, the first predicts the valuation of the input pattern, the second predicts the agent's amount of immobile stones and the third predicts the opponent's amount of immobile stones.

Degree of Freedom: The Degree of Freedom is defined as the amount of legal moves a player can make. This information is fed into the network, similar to *Immobility*.

Concentration: The concentration is defined in four steps, described in Winands et al. (2002). It describes how the stones are spread over the board. If all stones are centered around each other, the concentration is high. If all stones are disconnected from each other (highly separated), then the concentration is low. The concentration is calculated for both players and added to the state representation.

Biggest Cluster: The size of the biggest cluster is divided by 12, otherwise these values will have too much impact on the weights of the neural network. Each player starts with twelve stones, therefore the maximum values of the biggest clusters in the neural network will never be bigger than one.

4 Experiments

In this section we will describe the experimental setup, what opponents are used in the experiments and how the experiments are constructed.

4.1 Experimental Setup

The neural network used in the agent is a feedforward neural network. It has 64 or 66 input neurons, 50 hidden neurons as the second layer and one or three output neuron by default. As illustrated in section 3.7 the number of input and output neurons

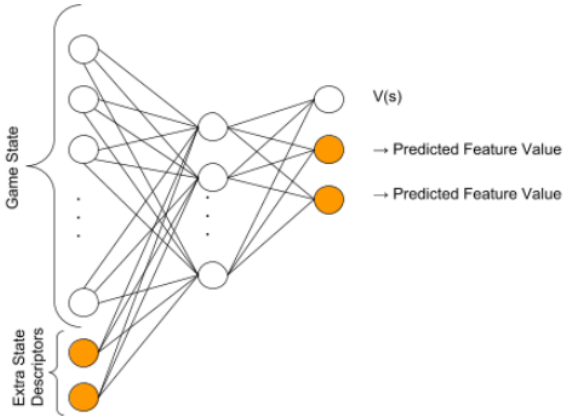


Figure 3.2: Two extra input neurons and output neurons are added. The first extra input neuron contains extra information about player 1 (learning agent). The second extra input neuron contains extra information about player 2 (opponent). The two extra output neurons predict the next state's values.

may increase if meta-features are involved. The hidden layer uses the activation function sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

The output layer uses activation function tanh:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})} \tag{4.2}$$

The function tanh is used to allow both positive and negative evaluations.

The weights of the hidden layer and output layer are initialized with a Xavier uniform distribution (Glorot and Bengio, 2010). The input of the network corresponds to all squares on the board, each neuron represents a square with a value of either 1, 0 or -1 (black piece, empty space and red piece respectively). As mentioned earlier, a win, lose and draw yield a reward of 1, -1 and 0 respectively. The discount-factor γ is set to 1.0. The exploration ϵ is decreased linearly from 1 to 0 in the first 80% of the game, after which it remains 0. The learning rate for the neural network is set to 0.001. The network is trained with batches of 500 games, where each batch is trained on three times in a random fashion. These batches consist of pairs of afterstates with a corresponding reward, which are used for determining and backpropagating the TD-error.

4.2 Opponents

The following opponents are used to test the agent's performance:

1) *Random agent:* It chooses a random possible move from all of its possible moves.

2) *COM:* It will execute an action based on these steps:

   1. Calculate the mean x and y values of all friendly stones $com(x, y)$
   2. Choose stone p, which is the most distant from $com(x, y)$.
   3. Order the moves of p according to which move will bring it closest to $com(x, y$.
   4. Choose a move from the sorted list according to the exponential decay function $y(x) = -2ln(x)$. Where x is a random value between 0 and 1. The value $y(x)$ is rounded down. This function ensures that the first move in the list will be picked with the highest probability.
   5. Execute the chosen move for the chosen stone.

   The COM agent includes stochasticity in its move choice, therefore it is a bench marking opponent with variation in its play.

3) *Self-Play generated opponent:* As an additional opponent, we took the resulting network from one run of 250,000 games trained with self-play and used this as the third opponent to test against.

Table 4.1 shows the performances of these opponents among each other.

Table 4.1: Performances of all players against each other. The performances are based on 100,000 test games.

SELF - COM	COM - RAND	SELF - RAND
0.88 - 0.12	1.0 - 0.00	1.00 - 0.00

4.3 Construction of Experiments

As a first investigation we train the network against each of the opponents described in section 4.2. For each opponent, the agent will play 250,000 games

which serve as training material. The probability of a player starting the game is 50%. As the starting positions of the players are identical, there is no need to change colour of stones. During these games, every 5,000 games the agent will play 1,000 test games against each of the opponents. In these test games there is no exploration and the agent will therefore use a greedy strategy. Despite being fully greedy, the agent is made partially stochastic as it takes its best three moves and executes them with a probability of 60%, 30% and 10% (best, second- and third- best moves respectively). This serves to include plenty of variation in the testing games. Noteworthy might be that the same goes for the self-play generated opponent described in the previous subsection. This will be done in 10 experiments in total where the MLP is randomly initialized each time. As such, we will be able to observe how the performance of the agent develops over time, which will allow us to determine which opponent allows the agent to perform best.

Secondly, we want to investigate whether the look-ahead function improves performance of the agent. The same experiment as described for the search of the best training opponent is used for this. However, during the 1,000 test games that occur every 5,000 games, the agent is armed with the look-ahead function. As such, we will be able to compare the results of these two experiments and observe whether the look-ahead affects the performance.

Lastly, in order to test whether the different input representations have an effect on the performance we, again, use the same experiment as described above. Now, during training, the network will be provided with the different input representations described in section 3.7 separately. That is, per feature there will be 10 runs for each training opponent. We can then compare the resulting performances to the initial investigation as we will do with the look-ahead investigation.

5 Results

Table 5.1 shows the resulting performances of training against each opponent: the random opponent (RAND), Center of Mass opponent (COM) and the generated self-play opponent (SELF). These are the average performances against each opponent over ten experimental runs. Additionally, Figure 5.1 shows the average performance development of training against each of the opponents over time, when tested against each opponent. This Figure uses the same data as used in Table 5.1.

Table 5.1: Performances of agents after training against each opponent. Each row represents a training opponent and each column represents a testing opponent. The performances showed are the average of ten experiments with the standard error.

Train vs.	COM	RAND	SELF
RAND	0.711 ±0.031	0.968 ±0.017	0.326 ±0.032
COM	0.799 ±0.025	0.974 ±0.018	0.304 ±0.018
Itself	**0.843 ±0.023**	**0.980 ±0.019**	**0.450 ±0.018**

Table 5.2: Performances of agents after training against each opponent using the look-ahead ability. Each row represents a training opponent and each column represents a testing opponent. The performances showed are the average of ten experiments with the standard error.

Train vs.	COM	RAND	SELF
RAND	0.722 ± 0.031	0.964 ± 0.018	0.370 ± 0.034
COM	0.850 ± 0.023	0.973 ± 0.018	0.321 ± 0.008
Itself	**0.854 ±0.023**	**0.980 ±0.018**	**0.503 ±0.014**

Table 5.3: Performances of agents after training against COM. Each column shows the performance of testing against an opponent averaged over ten experiments with the standard error for a specific input feature.

Meta features	COM	RAND	SELF
Normal	0.823 ±0.139	0.992 ±0.016	0.503 ±0.004
Immobility	0.879 ±0.162	0.994 ±0.012	0.505 ±0.008
DoF	**0.933 ±0.075**	**0.997 ±0.007**	0.509 ±0.012
Concentration	0.126 ±0.093	0.340 ±0.172	0.502 ±0.005
Biggest Cluster	0.821 ±0.167	0.986 ±0.038	**0.516 ±0.013**

In Table 5.2 we can view the resulting performance of training and testing against each opponent in the same fashion, where the agent uses the look-ahead function in the test games.

Tables 5.3, 5.4 and 5.5 show the average performance of each input feature being tested against various opponents. These performances are the average over 10 experimental runs. Figures A.2, A.3 and A.4 show the average performance development over time when trained against each opponent and tested against each opponent, with their respective input feature. After performing several t-tests on the best performing input feature per training opponent and test opponent compared with the normal representation, a few significant differences

Table 5.4: Performances of agents after training against RAND. Each column shows the performance of testing against an opponent averaged over ten experiments with the standard error for a specific input feature.

Meta features	COM	RAND	SELF
Normal	0.693 ±0.199	0.984 ±0.023	0.504 ±0.004
Immobility	**0.884 ±0.175**	0.999 ±0.001	0.505 ±0.005
DoF	0.880 ±0.109	0.999 ±0.004	0.508 ±0.008
Concentration	0.004 ±0.013	0.115 ±0.041	0.503 ±0.011
Biggest Cluster	0.855 ±0.074	**1.000 ±0.002**	**0.542 ±0.018**

Table 5.5: Performances of agents after training against SELF. Each column shows the performance of testing against an opponent averaged over ten experiments with the standard error for a specific input feature.

Meta features	COM	RAND	SELF
Normal	0.827 ±0.071	0.997 ±0.012	0.505 ±0.005
Immobility	0.838 ±0.101	**0.999 ±0.003**	0.505 ±0.005
DoF	0.826 ±0.108	0.998 ±0.008	0.507 ±0.006
Concentration	0.819 ±0.129	**0.999 ±0.003**	0.505 ±0.004
Biggest Cluster	**0.852 ±0.078**	**0.999 ±0.005**	**0.511 ±0.005**

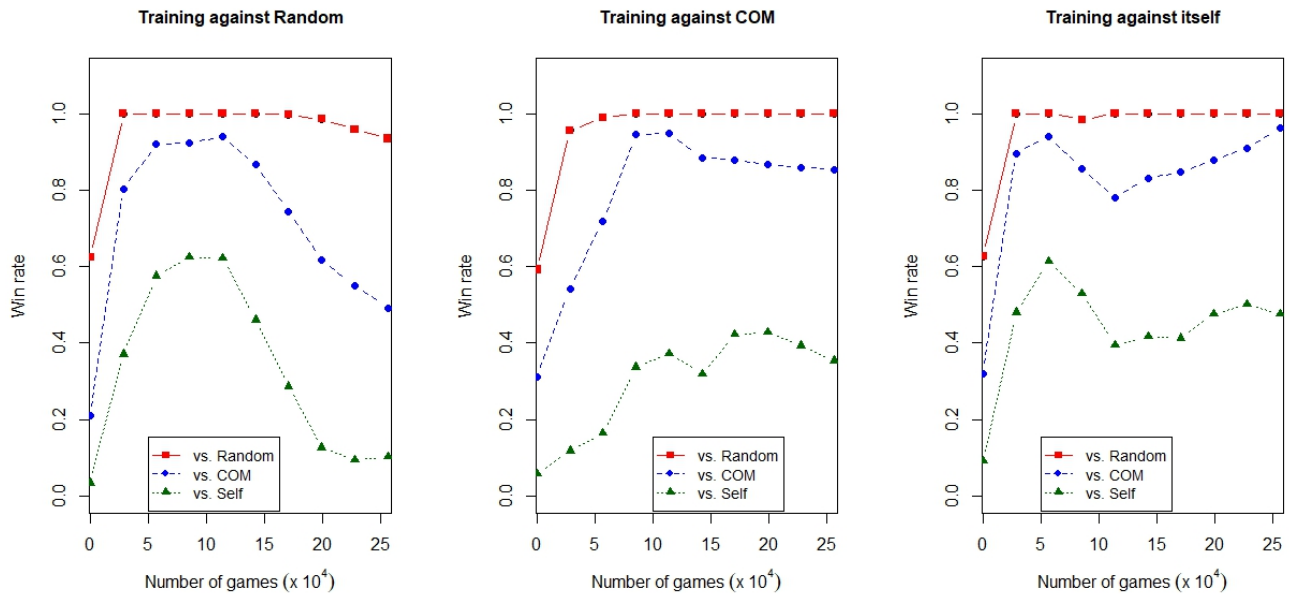
have been found. These values (Appendix A.3) can also be viewed in appendix A.

6 Discussion

In this section we make and discuss observations of the results obtained in the previous section, after which we answer the research questions using these observations.

6.1 Observations and Evaluations

- **Opponents** It turns out, as Table 5.1 indicates, that the agent trained against itself yields the best overall performance. The average win rate for self-play is higher against all opponents. We also observe that the overall performance of training against the Center of Mass (COM) opponent yields a higher win rate against the Random (RAND) and COM opponent in comparison to training against RAND. However, training against RAND seems to give a better overall performance against the self-play generated (Self) opponent than when training against COM. Despite the average being greater here, Figure 5.1 suggests that when looking at the resulting final performances, training against COM is significantly better than training against RAND. When training against RAND, the agent seems to be taking on a strategy which works well against all opponents, but over time, as convergence starts to occur, the resulting strategy seems to dramatically decrease performance. This likely occurs due to the fact that when training against a random opponent, the agent tends to take on a rather fixed strategy that only seems to be efficient against a random opponent. After converging to the thought to be optimal policy, if challenged with a more skillful opponent, this seems to lack the necessary generalization. The eventual strategies even appear to worsen the performance against RAND itself.
- **Look-Ahead** When we compare Table 5.1 and 5.2, we observe that if the agent uses the look-ahead ability against RAND, the performance seems to be slightly worse than without using it. As the look-ahead strategy assumes the opponent takes the best move, it logically follows that against an opponent that is purely random this has no effect and therefore may cause it to perform worse. However, when we compare the performances against COM and Self, we observe that with any training opponent the performance improves. The significant performance improvements are seen against Self when training against itself, and against COM when training against COM. As the former opponent does in fact use the assumed best moves of the look-ahead function (60% of the time at least), it logically follows there is a performance improvement. For testing against the COM opponent, the performance improvement can be explained by the fact that the moves this opponent executes are likely to be considered a best move according to the function. This, because, the strategy COM uses is a viable one in LOA, and could therefore be considered as rather good play.
- **Input Representation** When we take a look at tables 5.3, 5.4 and 5.5 we can see that input features generally improve performance when compared to the normal game representation. Some are statistically significant (appendix A.3). Figure A.2a shows that the degree of freedom improves performance for the agent



(a) **(b)** **(c)**

Figure 5.1: Average performance of training against the Random opponent, COM opponent and itself (a, b and c respectively). 250,000 games of training are used, in which the agent is tested against all opponents every 5,000 games for 1,000 test games.

that trains against COM and also tests against it. This might be due to the fact that, the COM strategy includes making a big cluster with few moves, since every stone is packed together. Therefore the learning agent can use this data for its own strategy. Interesting to note, performance is increased for training against RAND and testing against RAND for the 'biggest cluster' input representation. This is probably due to the strategy of RAND. It will only move stones randomly, therefore the agent can apply one single strategy, that is placing stones in the middle of the board. Therefore it is useful for the agent to know how big its own cluster is, in order to increase this cluster. The difference is not significant, since the performance quickly reaches near 100% against the random opponent. Additionally when it is tested against itself the agent performs better with the biggest cluster feature as well. Moreover training against itself with respective testing versus itself also yields a slight increase in performance with the biggest cluster input feature. The concentration feature either does not increase performance or it lowers it.

6.2 Answers to Research Questions

With the observations made in the previous section, we can now answer the research questions stated in the introduction:

- **Question** How do the performances of the learning agent trained against different opponents compare to each other?
  Answer Training with self-play results in the best learning for the agent considering the overall performance. Training against a random opponent appears to lead to the worst overall performance, whereas training against a fixed heuristic opponent results in better performance, yet still worse than training with self-play.

- **Question** How do the performances of the learning agent compare, when playing with and without a look-ahead function?
  Answer When the agent uses the provided look-ahead ability in testing games, it improves the performance against the fixed and self-play generated opponent, regardless of the training opponent. Against the random opponent it performs slightly worse, due to the lack of predictability of this opponent.

- **Question** How do the performances of the learning agent compare, when training with and without additional meta features of the game?
  Answer When the agent uses extra information to represent the state of the game, overall

performance increases when playing against a random opponent or a fixed opponent. Having additional information about the state of the game, does not yield a performance increase when training with self-play.

7 Conclusions

In this research we have compared several training opponents for a learning agent using temporal difference learning in the game Lines of Action: a random opponent, a fixed opponent using the so called Center of Mass heuristic, and the agent itself. Additionally, we investigated whether a look-ahead function, allowing the agent to perform a tree search for moves, improves performance, as well as whether feeding the network different input representations improves performance. We found that self-play serves as the best training opponent for the learning agent, and that enabling the agent to perform tree searches improves the performance against a non-random opponent. We also found that providing the network with meta-features improves the performance when training against a random or fixed opponent.
Future research, as the best training opponent is determined, might want to investigate whether the actual TD-Leaf algorithm used for training improves the resulting performance of the agent even further. Additionally, there are far more possibilities for input representations than noted in this paper which will be interesting to experiment with.

References

J. Baxter, A. Tridgell, and L. Weaver. Knightcap: A chess program that learns by combining TD(λ) with game-tree search.

J. Baxter, A. Tridgell, and L. Weaver. TDLeaf(lambda): Combining temporal difference learning with game-tree search. In *Proceedings of the Ninth Australian Conference on Neural Networks*, pages 168–172, 1998.

M. Buro. The evolution of strong othello programs. *Entertainment Computing*, pages 81–88, 2003.

Rich Caruana. Multitask learning. *Mach. Learn.*, 28:41–75, July 1997.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256, May 2010.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

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*, IJCAI'01, pages 529–534, 2001.

D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lnactot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T.P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–, 2017a.

D. Silver, Thomas Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T.P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and Shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017b.

R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

R.S. Sutton and A. Barto. Reinforcement learning: An introduction. *MIT press*, 1998.

G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

M. van der Ree and M. Wiering. Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play. In *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 108–115, 2013.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

M.H.M. Winands. Analysis and implementation of lines of action. Master's thesis, 2000.

M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. The quad heuristic in lines of action. 2001.

M.H.M. Winands, L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Temporal difference learning and the Neural MoveMap heuristic in the game of Lines of Action. In *Proceedings of 3rd International Conference on Intelligent Games and Simulation (GAME-ON 2002)*, pages 99–103. SCS Europe Bvba, Ghent, Belgium, 2002.

A Appendix

A.1 Significant Statistics of Opponents

Training against itself vs. training against random shows a significant difference when testing against COM and Self in favour of self-play ($t(93) = -3.42, p < .05$; $t(78) = -3.40, p < .05$, respectively).

Training against COM vs. training against random shows a significant difference when testing against COM in favour of training against COM ($t(96) = -2.20, p < .05$).

Training against itself vs. training against COM shows a significant difference when testing against Self ($t(100) = -5.89, p < 0.05$).

A.2 Significant Statistics of Look-ahead

Training against COM shows significant improvement when tested against COM ($t(99) = -2.00, p < 0.05$) compared to without look-ahead.

Training against itself shows significant improvement when tested against Self ($t(100) = -2.03, p < 0.05$) compared to without look-ahead.

A.3 Significant Statistics of Input Representations

Degree of freedom increased performance when trained against COM and tested against COM, $t(75) = -4.91, p < .05$.

Biggest cluster is significant for training against COM, while testing against SELF $t(59) = -6.47, p < .05$.

Immobility seems to improve performance when training against RAND and tested vs COM $t(96) = -5.10, p < .05$.

Biggest cluster improves performance when trained on RAND and tested against RAND $t(49) = -4.92, p < .05$.

Biggest cluster improves performance when trained on RAND and tested on SELF $t(55) = -14.6, p < .05$.

Biggest cluster improves performance when trained against SELF and tested against SELF $t(98) = -5.87, p < .05$.

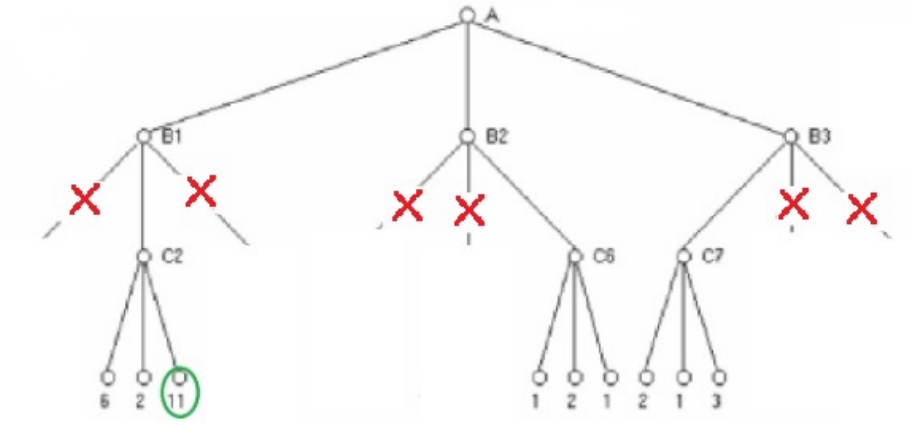
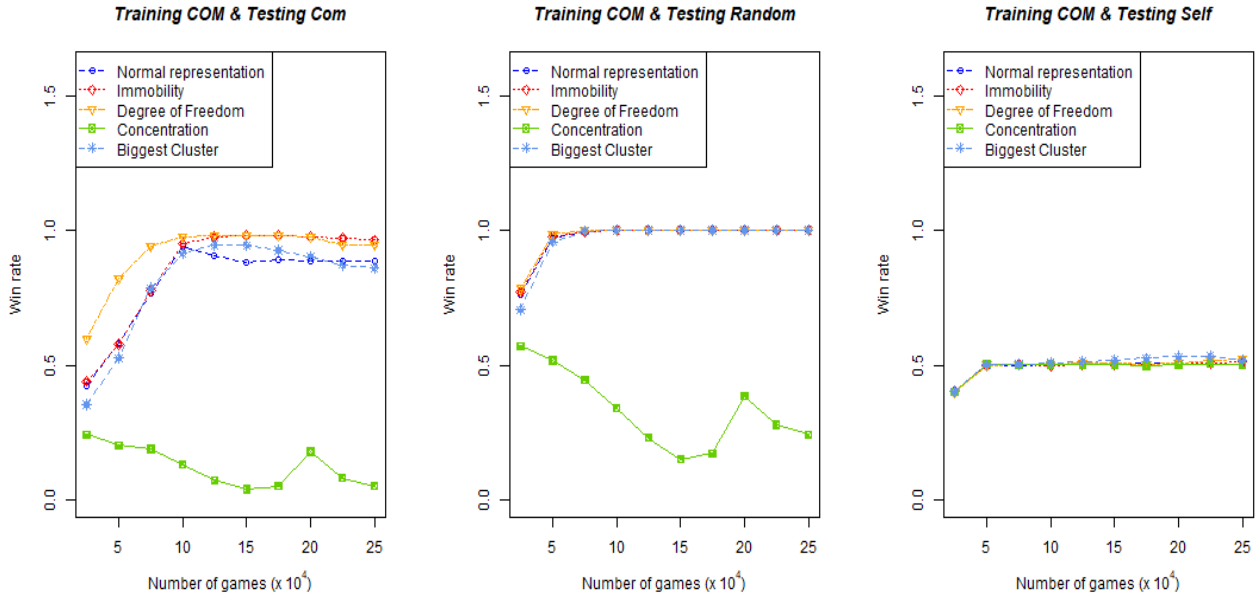
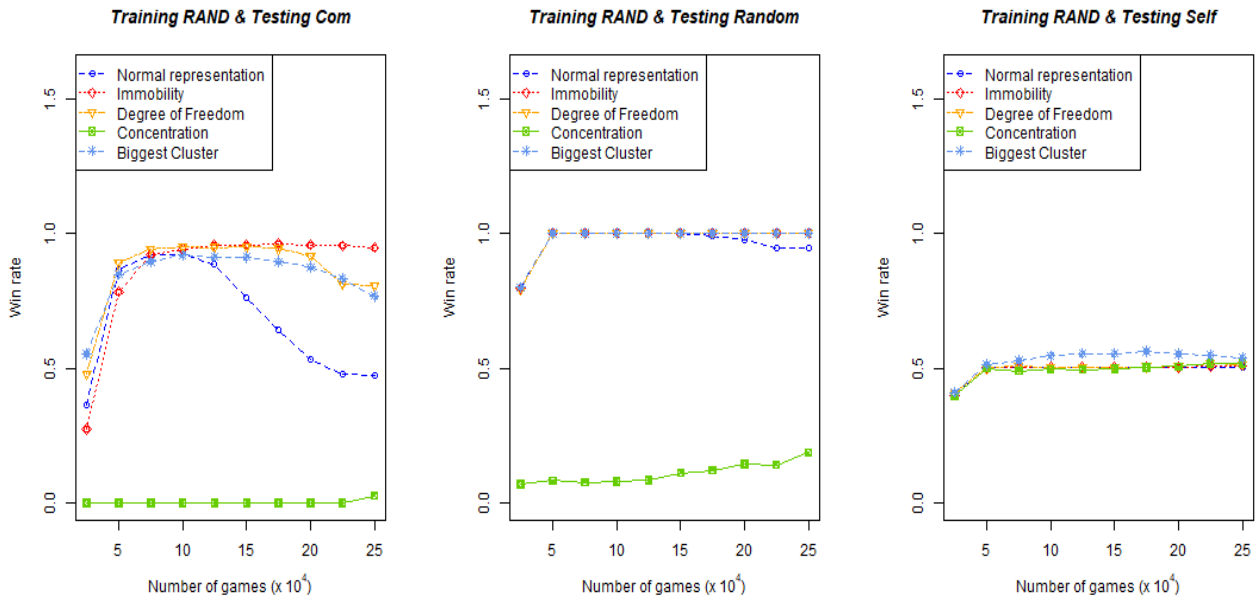


Figure A.1: This figure illustrates look-ahead function used in this paper. The B-nodes represent the three moves with the highest valuation in state A. The C-nodes are possible moves of the opponent in the resulting states, of which only the best ones remain, the other ones are pruned. Finally, we have the leaf-nodes which represent the possible moves in the resulting states after the opponent moves, of which the leaf with the highest value is determined. In this case the leaf with value 11 is highest, meaning that action B1 will be executed.



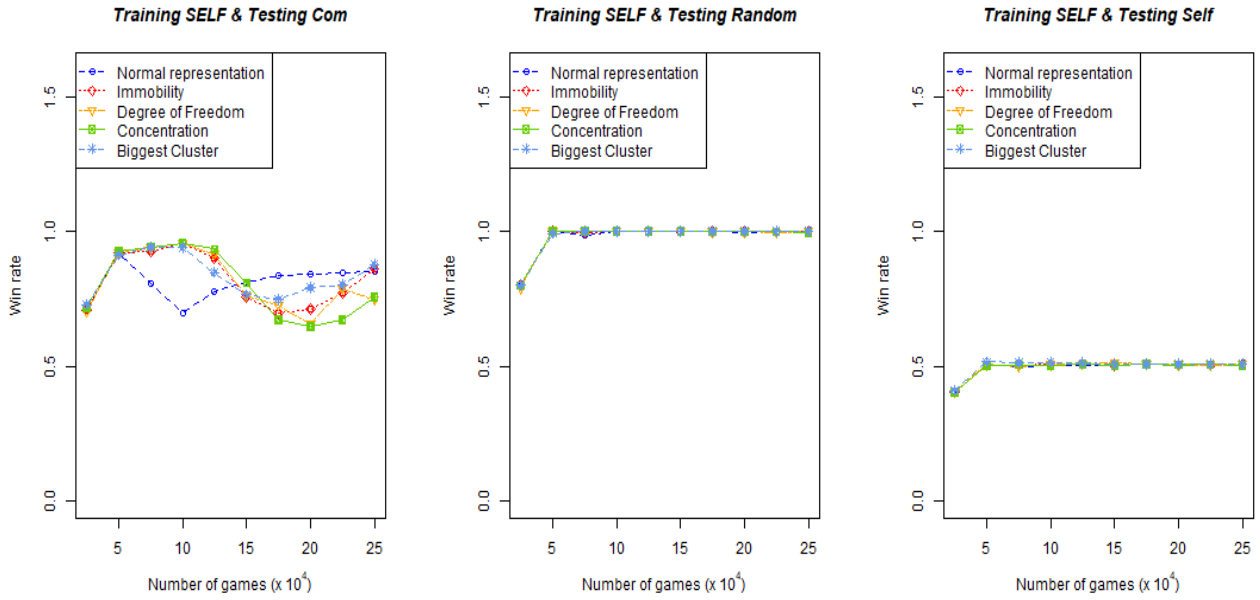
(a) **(b)** **(c)**

Figure A.2: Average performance of training against a centre of mass opponent, and tested against various opponents, while playing 250,000 games. Every 5,000 games there are 1,000 test games.



(a) (b) (c)

Figure A.3: Average performance of training against a random opponent, and tested against various opponents, while playing 250,000 games. Every 5,000 games there are 1,000 test games.



(a) (b) (c)

Figure A.4: Average performance of training against itself, and tested against various opponents, while playing 250,000 games. Every 5,000 games there are 1,000 test games.

8 Division of Work

8.1 Implementations

Table 8.1: Contributions to implementations

Implementation of:	Contributor(s)
Game rules/mechanics	Quintin and Remo
Multi-Layer perceptron	Quintin and Remo
Centre of Mass Opponent	Quintin
TD(0)	Remo
Batch-training	Quintin and Remo
Self-play	Remo
Look-ahead	Remo
Input features	Quintin

8.2 Experiments

Table 8.2: Contributions to each experiment

Experiment	Contributor(s)
Opponent search	Remo
Look-ahead	Remo
Input features	Quintin

8.3 Writing

Table 8.3: Contributions to each (sub)section

Section	Contributor(s)
Abstract	Remo and Quintin
Introduction	Remo
Lines of Action	Quintin
Reinforcement Learning	Remo
Temporal Difference Learning	Remo
Function Approximator	Quintin
Application to Lines of Action	Quintin
Learning from Opponents and Self-Play	Remo
Look-ahead	Remo
Input Representation	Quintin
Experimental Setup	Quintin
Opponents	Quintin
Construction of Experiments	Remo
Results Opponents and Look-ahead	Remo
Results Input Features	Quintin
Discussion Opponents and Look-ahead	Remo
Discussion Input Features	Quintin
Conclusion	Remo
References	Quintin