Neural-Fitted Temporal Difference Learning to Learn to Play Connect Four

(Bachelor Thesis)

H. de Haan, s2068125, heindehaan@gmail.com,
M.A. Wiering*,

July 31, 2013

Abstract

In this thesis, neural-fitted temporal difference learning, a form of reinforcement learning, is used to learn to play the game of Connect Four. Seven different artificial players are compared, using five different neural networks. While the first network only uses the basic board state as input, the larger ones also use specific features: rows of two, three and four in the different rows of the board state. It is shown that these features dramatically improve the performance of the agent. Furthermore, two different exploration strategies are used: Boltzmann with constant temperature and ϵ-greedy. The results show that ϵ-greedy gives the most stable result. Finally, the smallest network was given the same number of hidden nodes as the largest network, showing that adding hidden nodes does not improve the score of the system.

1 Introduction

Letting a computer learn to play the game of Connect Four can be done in multiple ways. For example, Allis (1988) used a set of fixed rules to program the behavior of VICTOR, which as a result played a perfect game of Connect Four. Given that VICTOR has the first move, it will always win, and given that VICTOR's opponent begins and does not start in the middle column, VICTOR can always ensure at least a draw (Allis (1988)). However, a completely different approach to learning to play the game has also been proven to work: using neural networks. The neural network gives values to the different actions the machine can take, after which the move with the highest value is chosen. This is called the connectionist approach and is used by Schneider and Garcia Rosa (2002). However, Schneider and Garcia Rosa (2002) used a supervised learning algorithm. The approach used in the research of this paper is also a connectionist approach, but it uses a form of reinforcement learning to learn to play Connect Four. Specifically, it uses neural-fitted temporal difference learning (Van den Dries and Wiering (2012), Sutton (1988)). The idea is to let the artificial player play a large number of games (100.000) against itself, assuming both the role of the white and the black player. Having played fifty games, the system uses all the stored board positions to learn an evaluation function which is used for move selection. The learning agent is tested against two simple test agents: one of which plays purely random moves, while the other plays random moves unless it can win in just one move, in which case it does so, and unless its opponent can win in one move, in which case the test agent tries to block that possibility. Seven different learning players are compared for this research. The first five differ in the input size of the neural network: the first network only uses the basic board state as input, while the larger networks also use specific features of the board state: rows of two, three and four in the different rows of the board state. The sixth player uses a different exploration method than the first five: Boltzmann exploration instead of ϵ-greedy. Finally, the seventh player is the same as the first, but uses the same number of hidden nodes as the largest

*University of Groningen, Department of Artificial Intelligence

network. All other players use different numbers of hidden nodes, so this seventh system was created to test whether the number of hidden nodes influences the performance.
The outline of this paper is as follows: first, general aspects of Connect Four are discussed (such as the rules of the game). After that, the learning algorithm of this research, neural-fitted temporal difference learning, will be discussed in the context of the general concept of reinforcement learning. The performed experiments and results will then be discussed, after which a conclusion follows.

2 Connect Four

2.1 Rules of the Game

Connect Four is a two player, zero-sum game. The game is played on a vertical board with seven columns and six rows. Each player has his or her own color: white or black. White begins. The players take alternating turns, and on each turn the current player drops a disc of his or her color in one of the columns. The disc will fall until it reaches the bottom or the top of the last disc dropped in that column. The objective of each participant is to create a vertical, horizontal or diagonal contiguous row of (at least) four pieces of his or her color before the opponent does so. The game ends in a draw if all the fields of the board are taken and both players didn't succeed in making a row of four discs.
One can immediately see that simply making a row of three of the same pieces to make a row of four in the future isn't really good: the opponent can easily block such a row. It's better to have a row of three of your pieces which can be expanded to two different rows of four in your next move: your opponent can only block one of the opportunities, so you will win in your next move. A good artificial connect four player must take this into account.

2.2 Complexity of the Game

Of course, an important property of each game is its complexity: the number of possible states the game can be in. It's easy to see that, in the case of Connect Four, each field of the board can be in three different states: it can be empty and it can contain a white or a black piece. It follows that the upper bound of the number of states is 3^{42} (Allis (1988)). Of course, this upper bound contains many illegal states. For example, a state in which the bottom field of a column is empty but a higher field is filled can never occur with legal play (Allis (1988)). Furthermore, you can't have a state with 10 white discs and 4 black discs: because of the alternating turns, there are always as many white as black pieces in the game unless white just played, in which case there is one more white disc. Considering all limitations, Allis (1988) found an upper bound of $7.3 * 10^{13}$. Clearly, this complexity is too large for a simple brute force algorithm on a standard personal computer. In the next section, a method is described to handle this complexity.

2.3 Related Work

In Allis (1988), a knowledge-based approach to playing Connect Four is described. The idea is fundamentally different from the approach given in this thesis, as later is shown. Allis (1988) gives a set of rules to program the behavior of VICTOR, its Connect Four playing program. Using this set of rules, VICTOR can always win with white and at least draw with black given that white doesn't start in the middle column (Allis (1988)).

3 Reinforcement Learning

In this section, a short introduction to the concept of reinforcement learning will be given. The general idea of reinforcement learning is to let an agent perform a specific task without specific instructions on how to do that task; the supervisor (the human programmer) only gives the agent punishments and rewards for respectively bad and good performances of the agent (Kaelbling, Littman, and Moore (1996)). It's important to note that, in Connect Four, a reward (positive or negative) is only given at the end of the game: a positive reward for a win, an equally big but negative one (a punishment) for a loss and a neutral one for a draw. More formally, we have a model that consists of the following parts: a set of discrete environment states, a set of actions that the agent (in this case the artificial Connect Four player) can perform and a set of reinforcement signals (in this case {-1,0,1}). Performing a certain action brings

the agent from one environment state to another, possibly while receiving a reward. The goal of the agent is to find a policy (a policy maps states to actions) that maximizes the expected long-term reward (Kaelbling et al. (1996)). In Connect Four, the long-term reward is simply the reward at the end of the game. So, the agent has to find a policy that maps Connect Four board states to Connect Four actions so that the chance of winning the game is maximized. To find the optimal policy, value-function based reinforcement learning makes use of value functions: such a function receives updates from experiences of the agent and in that way it summarizes the results of those experiences (Van den Dries and Wiering (2012)). The value function can then be used to select actions: it gives a value to all possible follow states given a certain state. The follow state with the highest value is chosen. The optimal policy then is the policy which leads to the highest state-value in all states (Van den Dries and Wiering (2012)). So, instead of planning a sequence of actions, reinforcement learning is about finding policies mapping states onto actions in such a way that the expected outcomes are the desired outcomes (Wiering and van Otterlo (2012)).

3.1 The Function Approximator

The artificial player in this project is trained by playing a large number of games against itself; therefore, it plays both black and white. However, the function approximator cannot use these colors; it just has to know which pieces are the current player's pieces and which ones are not (the same approximator is used for the white and for the black player, which are the same player). Therefore, in the future, the "players" will be called current player and opponent.

As said earlier, value-function based reinforcement learning uses a value function. In this project the value function is learned by a fully connected feed-forward neural network with three layers (the function approximator). Fully connected means that all input nodes are connected with all hidden nodes, and all hidden nodes are connected with all output nodes. No direct connections exist between input and output nodes. Various networks were created and tested. The first one consists of 42 input nodes: each one corresponding with a field of the Connect Four board. Each input can be in three states: -1, meaning the corresponding field is occupied by the opponent, 0, meaning that the field is unoccupied, and 1, corresponding with a field filled with a disc of the current player. As later is shown, this basic network yields poor results after training. To make it perform better, extra features were added to the network. The first and obvious feature is an input node which tells if the board state contains a row of four discs of the current players color. This node can be in two states, 0 if no such row exists and 1 if one or more rows of four pieces of the current players color exist. In other words, this node tells the system if a board state is a winning position for the current player. No such node exists for a row of four discs of the opponent's color, because such a node would always output a 0: if the opponent's color has a row of four discs, the game ends immediately. Furthermore, the current player cannot make a row of four discs of the opponent's color. So, for move selection, such a node would be completely useless. What is, as later is shown, not useless is nodes corresponding with nearly completed rows of 4. If somewhere in the board state there are three discs of the same color which only need one more disc to make a complete row of four, a specific node value is increased by 1. Of course, different nodes exist to distinguish between the different player's nearly completed rows. Also, different nodes exist to take into account the height of the nearly completed row (the height of the empty field in the nearly completed row is used). Finally, there are different nodes to differentiate between horizontal, vertical and two kinds of diagonal nearly completed rows. In total, the nearly completed rows are coded with twenty-one input nodes. The same thing is done for rows of two discs of the same color. Here, for vertical and diagonal rows, the height of the lowest disc is used. The neural network with al these features thus knows how many rows of two and how many nearly completed rows of four discs there are in each row of the board state. Finally, an input node exists to count the total number of nearly completed rows of the current player's color and one that does the same for the opponent's color.

The neural network uses one hidden layer, which consists of half the number of nodes of the input layer. The output layer has only one node, which gives the value of a certain board position.

3.2 Neural-Fitted Temporal Difference Learning

In Van den Dries and Wiering (2012), the variant of reinforcement learning used in this research is described: neural-fitted temporal difference learning. To understand this algorithm, it's helpful first to look at the general concept of temporal difference learning. According to Sutton (1988), temporal difference learning updates the state value function after the agent makes a transition from one state (s_t) to another state (s_{t+1}), in which it receives a certain reward (r_t):

$$V(s_t) := V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)) \quad (3.1)$$

$0 < \alpha \leq 1$ is the learning rate, and $0 < \gamma \leq 1$ is the discount factor. Basically, the discount factor says how much the value of one state should be adapted to the value of the next. The state value function that selects actions looks like this:

$$\pi(s) = arg\ \max_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V(s')) \quad (3.2)$$

Here, $P(s,a,s')$ is a function that maps states (s) and actions (a) to a probability distribution of successor states s' (Van den Dries and Wiering (2012)). Furthermore, $R(s,a,s')$ gives the average reward for going from state s to s' using action a. As said before, in Connect Four (and many other games) there is only a reward at the end of the game (correlating to a win, a loss or a draw). Therefore, not in every state transition there's a reward. Also (Van den Dries and Wiering (2012)), there are two players playing against each other, and the system needs to learn from both the white and the black player. Because of this, the learning rule (3.1) needs to be changed. The idea is to update the value of a certain game state with white on move using the value of the next state with white on move. Of course, the same can be done for black. Let $x = (x_1, ..., x_m)$ be the set of game states after black has played. For white, let $y = (y_1, ..., y_m)$ be such a set. Note that we're looking at state-values of states after a player has made a move, which makes sense because these values can be used later to select moves. After a training game is played, it's straightforward first to look at the last game states, since the values to assign to these states are easy (Van den Dries and Wiering (2012)):

$$V(x_m) = r_{x_m} \quad (3.3)$$

$$V(y_m) = r_{y_m} \quad (3.4)$$

x_m is simply the reward given to the black player, and y_m the one for the white player. So, if white won, $y_m = 1$ and $x_m = -1$. For all the other state values, the following equation is used for $s = x$ and $s = y$ (Van den Dries and Wiering (2012)):

$$V(s_t) = \gamma V(s_{t+1}) \quad (3.5)$$

So, for each state, the value of the next state after which the same player played is used. The value of that next state is given by the function approximator.

Neural-fitted temporal difference learning (Van den Dries and Wiering (2012)) is a batch algorithm, meaning the system first plays a number of games and then the training occurs. After that, new games can be played and new training can begin, etcetera. The state - state-value patterns given by the previous equations are used to train the neural network (the function approximator). The interesting part is that after training on a state and its value, the neural network changes, so it will give another value to the same state (Van den Dries and Wiering (2012)). Therefore, the same state can be used repeatedly for training. The algorithm thus is as follows (Van den Dries and Wiering (2012)):

1. The system plays a number of games, which are all stored together with their results.
2. Repeat the following a number of times: use the function approximator to get the target values of all the board-states of all the games, and train the function approximator with these states and their assigned values.

3.3 Exploration Methods

An important aspect of each reinforcement learning implementation is the method of exploration. The point is that, during training games, it's not wise to always select the action that is best according to the current function approximator (this is what's done by so called greedy algorithms): actions with low values may give rise to opportunities in the future (Sutton and Barto (1998)). A simple exploration method is ϵ-greedy, which, with a certain probability, chooses a random action, and otherwise chooses the one which is given the highest value by the function approximator (Vermorel and

Mohri (2005)). A problem with this approach is that the exploration is random: if the learning player doesn't choose the action it thinks is best, but a random action, all actions (except of course the apparent best action) are equally likely to be chosen (Kaelbling et al. (1996)). A more sophisticated exploration method is called Boltzmann exploration; the idea is to assign likelihoods to each action, with higher likelihoods for better actions. The probability for each action is calculated as follows (Kaelbling et al. (1996)):

$$P(a) = \frac{e^{\frac{V(a)}{T}}}{\sum_{a' \in A^e} \frac{V(a')}{T}} \tag{3.6}$$

T is the "temperature", which is set to a stable 0.5 for this research. $V(a)$ is the expected value of the follow state given that action a is performed. Both exploration strategies are used and tested in this research.

3.4 Related Work

A very succesful artificial game player that used temporal difference learning was TD-Gammon (Tesauro (1994)), which, as the name suggests, played backgammon. Using only a raw board state as input at the beginning of learning, TD-Gammon learned to play at a strong intermediate level. However, adding features to the input representation made the sytem perform even better, at a level close to the world's best human players.

Van den Dries and Wiering (2012) used neural-fitted temporal difference learning in an Othello-playing program. While also using a neural network with three layers, the main difference with the network in this paper is that it is structured, meaning a number of links between the input and the hidden layer are removed. This decreases the number of parameters that need to be learned, which is useful for a game as complex as Othello (Van den Dries and Wiering (2012)).

In van der Ree and Wiering (2013), it is argued that, for learning the game of Othello using temporal difference learning, learning from playing against yourself works better than learning from playing against a fixed opponent.

4 Experiments and Results

As discussed earlier, five different artificial Connect Four players were created, which differ in the neural network they use. The first player uses a network which only uses the basic board state as its input. It obviously has 42 inputs and will therefore be called NET42. The second system also has a node corresponding to whether or not the board state is a winning state for the learning player. Because it has one extra node compared to the first network, it has 43 nodes and will be called NET43. The third network has all the previous inputs plus two nodes that count the number of nearly completed rows of four in the six columns of the system itself and its opponent, respectively. The empty field of horizontal rows can be in every board row, so that gives six extra input nodes (one for every row in the board). For the two kinds of diagonal rows, the same holds; twelve inputs exist for those rows. The empty field in vertical rows can only be in the top three rows of the board, because one can never put a disc beneath other discs. This makes a total of 21 extra inputs per color, so 42 overall. Taking into account the 43 input nodes that already existed, this network has 85 input nodes and is called NET85. The fourth neural network also has inputs for the number of rows of two discs of the learning players color in the different rows of the board, and the same for rows of two of the opponent. The position of the lowest of the two discs is used; in horizontal rows of two, the "lowest" can be in all six rows of the board. For the two kinds of diagonal rows and for vertical rows, the lowest disc obviously can't occur in the top row. All other rows are available. So, taking into account the two possible colors, the fourth system has 42 extra input nodes for a total of 127 input nodes (NET127). Finally, the fifth and largest network has all the previous inputs, plus two inputs that count the total number of nearly completed rows of four of the learning player and of its opponent, respectively. Of course, this system is called NET129. The number of hidden nodes in each network is simply half of the number of input nodes. All these learning players use ϵ-greedy exploration. To compare ϵ-greedy with Boltzmann exploration, the largest network was tested in comparison with Boltzmann exploration. The resulting system is named BOLTZMANN. Finally,

Table 1: Overview of the different networks.

System	# Hidden Nodes	Description
NET42	21	Basic board state
NET43	21	Rows of four
NET85	42	Nearly completed rows of four in each board row
NET127	63	Rows of two
NET129	64	Total number of nearly completed rows of four
BOLTZMANN	64	Boltzmann
NET42a	64	Extra hidden nodes

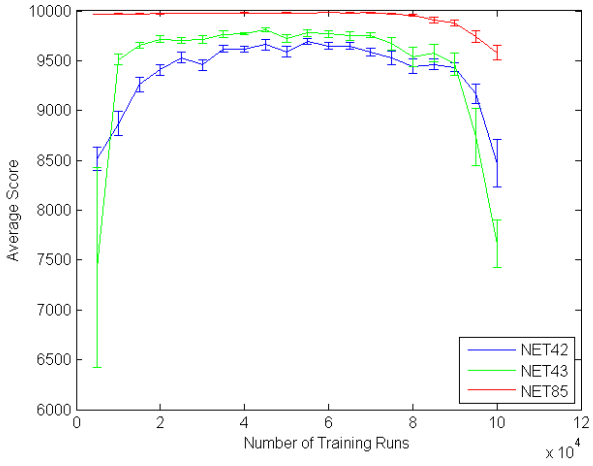

Figure 1: NET42, NET43 and NET85 vs. RANDOM

to decide whether the number of hidden nodes influences performance, NET42a was created. This player is the same as NET42, except for the fact that it uses the same number of hidden nodes as NET129: 64. An overview of the seven learning players is given in table 1.

To test the different sytems, two test agents were built. The first agent is a random agent; at each turn, this agent chooses its move at random. Let's call this player RANDOM. The second agent has two rules (in descending priority): if, somewhere in the Connect Four board, he can win in one move, he does so. Furthermore, if his opponent has that possibility in the opponent's next move, he blocks that possibility. Of course, in a situation where his opponent has two possibilities to win in his next move, the test agent cannot block both and may lose anyway. This agent will be called RANDOM2. The experimental setup was as follows: each artificial player (NET42, NET43, NET85, NET127, NET129, BOLTZMANN) played 100.000 games against themselves to train their weights. After each 5000 training runs, they were tested by playing 10.000 games against RANDOM and 10.000 games against RANDOM2. Winning a match is worth 1 point, drawing 0.5 points and losing, of course, 0 points. This process was repeated ten times for each learning agent. Testing after each 5000 training runs was done to show the learning curve of the learning agents; only testing at the end of the training runs will just show the end results. Furthermore, as later is shown, the learning curves have a peak somewhere, after which performance drops. Because of this, the best results are not per se at the end of the 100.000 training runs. The learning rate was set to 0.008 for all experiments. For systems using ϵ-greedy, ϵ always began at 0.42 and was linearly lowered to end at zero at the end of the training runs. Systems using Boltzmann-exploration used a temperature of 0.5 which did not change during training. The highest average scores of all agents against RANDOM and RANDOM2 is given in table 2. The highest average score is simply the highest point in the learning curve of an agent. Table 2 also shows the standard error of the scores. Finally (and most importantly), two-sample t-tests (not assuming equal variances) were performed to see whether each player performed better than its predecessor. So, for example, NET43 was compared to NET42, and NET85 with NET43, etcetera. BOLTZMANN was compared to NET129, and NET42a was, of course, compared to NET42.

Figure 1 shows the learning curves of the smallest three learning players, tested against RANDOM, with standard error bars. The results seem to show improvement with the growth of the network: NET43 performs at its highest point significantly better than NET42 at its highest ($p < 0.01$), and NET85 performs even better ($p < 0.001$). With a 9996.3 score against RANDOM, NET127 is significantly better ($p < 0.001$) than NET85.

Table 2: Best average scores of the different networks, with their standard errors. Each system was compared with its predecessor in a t-test. BOLTZMANN was compared with NET129 and NET42a with NET42.

	RANDOM	Standard Error	P-Value	RANDOM2	Standard Error	P-Value
NET42	9693.5	30.09		6658.1	156.12	
NET43	9808.5	22.47	<0.01	6445.1	180.33	0.38
NET85	9982.9	1.28	<0.001	8887.7	57.97	<0.001
NET127	9996.3	0.66	<0.001	9190.7	31.13	<0.005
NET129	9997.4	0.49	0.20	9266.9	38.76	0.14
BOLTZMANN	9998.6	0.39	<0.1	8949.8	38.70	<0.001
NET42a	8691.9	50.70	<0.001	4323.4	166.58	<0.001

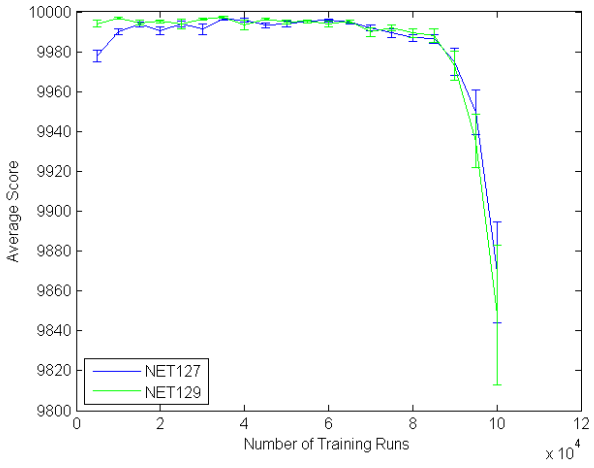

Figure 2: NET127 and NET129 vs. RANDOM

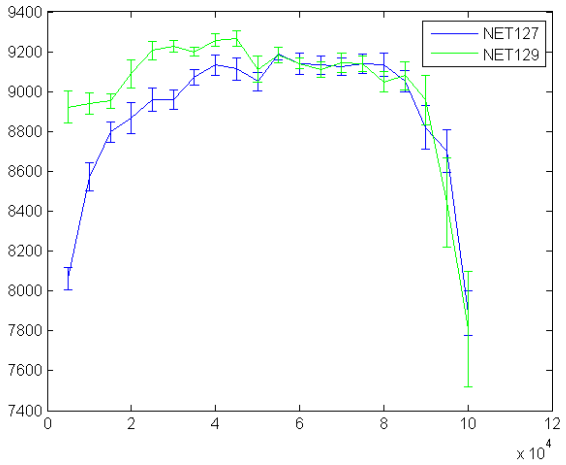

Figure 4: NET127 and NET129 vs. RANDOM2

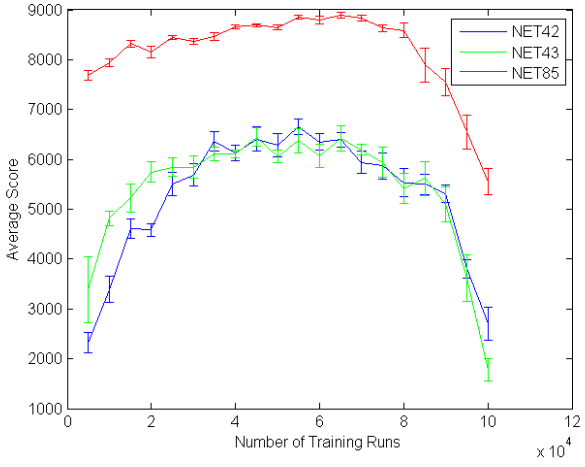

Figure 3: NET42, NET43 and NET85 vs. RANDOM2

However, the two extra counters in NET129 did not improve the score against RANDOM. In Figure 2, the learning curves of NET127 and NET129 look very similar. Looking at the scores against RANDOM2, NET43 and NET42 don't perform significantly different. In Figure 3, it is shown that their learning curves are very similar. NET85 is however better than NET43 ($p < 0.001$), which is clearly visible comparing their learning curves. Furthermore, with $p < 0.005$, NET127 has a higher score than NET85. Again, NET129 does not perform significantly better than NET127. However, looking at Figure 4 it is shown that NET129 does start out quite a bit higher than NET127. The highest average scores of the seven learning agents against RANDOM2 are also shown in table 2.

Compared to ϵ-greedy, Boltzmann-exploration results in a similar learning curve when using RANDOM as test agent: the curve of BOLTZMANN when tested against RANDOM (Figure 5) starts out high, does not go up much and ends lower than it started. At its highest point, BOLTZMANN scores higher against RANDOM than NET129 ($p < 0.1$). Comparing BOLTZMANN tested against RANDOM2 with NET129 (Figure 6, Figure 4) shows that BOLTZMANN performs less learning, having a less steep learning curve than NET129. Using ϵ-greedy, NET129 performs better than BOLTZMANN ($p < 0.001$).
NET42a performs significantly worse than NET42, both when tested against RANDOM and when tested against RANDOM2 ($p < 0.001$ for both). From this, it can be concluded that simply increasing the number of hidden nodes does not improve the score of an agent. The differences in the listed scores must be caused by the added features.
Note that the scores of all the artificial players using ϵ-greedy decrease a lot after about 50.000 training runs. To test whether this is caused by ϵ ending to low (at zero), NET129 was again tested against RANDOM and RANDOM2 as before, but with ϵ ending at 0.20. The results are shown in Figure 7. Comparing this figure with Figure 2 and Figure 4, it is clear that the new range of values for ϵ causes the scores to decrease much less towards the end of the 100.000 training runs.

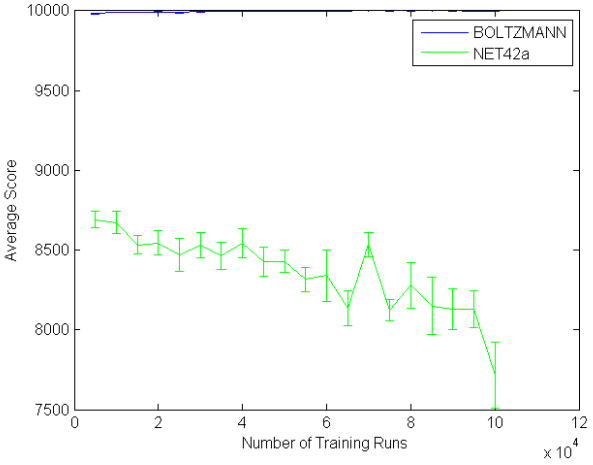

Figure 5: BOLTZMANN and NET42a vs. RANDOM

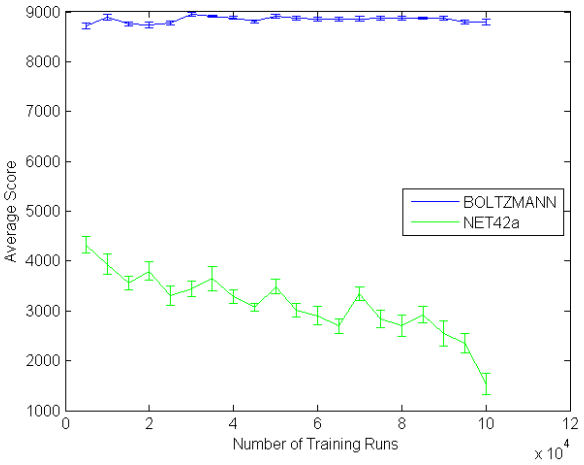

Figure 6: BOLTZMANN and NET42a vs. RANDOM2

5 Conclusion and Future Work

In this thesis, seven learning Connect Four players were compared with each other, in test runs against a random agent and a sophisticated random agent. They differed among each other with respect to the size of the input layer of the network: the first one only used the basic board state as input, while the larger ones used specific features of the board state to perform better. Also, ϵ-greedy was compared to Boltzmann exploration, and, finally, an agent was created that was the same as the smallest learning agent but had more hidden nodes (the same number as the largest learning player). The results showed that the added features all made

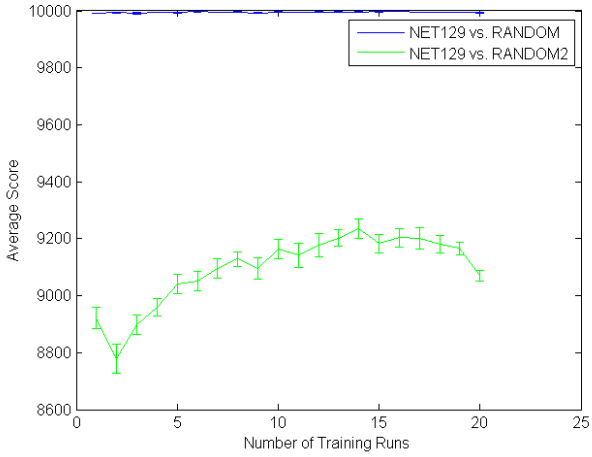

Figure 7: NET129 vs. RANDOM and NET129 vs. RANDOM2, with ϵ ending at 0.20

the system perform better against the random agent except for the extra two features in the network with the largest input layer. Against the sophisticated random agent, both the second and the largest system did perform better than their predecessors, but all other learning players did. Boltzmann and ϵ-greedy showed different learning curves, with ϵ-greedy performing better against the sophisticated random agent and equally against the random agent. Simply adding hidden nodes did not improve the score of the smallest learning player, indicating that the differences in performance were caused by the added features. Finally, it was shown that the decrease in the scores of the players using ϵ-greedy was caused by the fact that ϵ ended at zero: 0.20 as lowest value for ϵ proved to cause the scores of NET129 to decrease much less towards the end of the training runs.

One obvious possible expansion of the used neural networks is adding the exact position of the different features, instead of only using their row of the board. Furthermore, extra features could be added to the network to make it perform better; maybe the distribution of the different pieces over the board. However, both ideas will of course dramatically increase the size of the neural network, and thus make it less time-efficient.

References

V. Allis. A knowledge-based approach of connect-four. Master's thesis, Vrije Universiteit Amsterdam, 1988.

L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

M.O. Schneider and J.L. Garcia Rosa. Neural connect 4 - a connectionist approach to the game. In *VII Brazilian Symposium on Neural Networks*, pages 236–241, 2002.

R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1): 9–44, 1988.

R.S. Sutton and A.G. Barto. *Introduction to Reinforcement Learning*. MIT Press Cambridge, 1998.

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

S. Van den Dries and M.A. Wiering. Neural-fitted td-leaf learning for playing othello with structured neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 23:1701–1713, 2012.

M. van der Ree and M.A. Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. In *IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2013.

J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *European Conference on Machine Learning*, pages 437–448. Springer, 2005.

M.A. Wiering and M. van Otterlo. *Reinforcement Learning: State-of-the-Art*. Springer, 2012.