Learning Othello using Neuroevolution and Temporal Difference Learning

Master Thesis Agents and Computational Intelligence

M. Vuurboom

January 24, 2008

Universiteit Utrecht

Learning Othello using Neuroevolution and Temporal Difference Learning

Michiel Vuurboom

Master Thesis
Utrecht, January 24, 2008

University of Utrecht, the Netherlands
Department of Information and Computing Sciences
Master Agents and Computational Intelligence

Supervisor:
Dr. M.A. Wiering (University of Utrecht)

Committee:
Dr. M.A. Wiering (University of Utrecht)
Dr. F.P.M. Dignum (University of Utrecht)

Preface

In 2004 I started my master Agents & Computational Intelligence after finishing a bachelor in Information and Communication Technology in Enschede. I have always been interested in machine learning and of course in games. During this master I took all courses about learning and about games, because those are my favourite subjects in the field of Artificial Intelligence.
After finishing all my courses I started my master thesis in April 2006 and tried to do research on several aspects of gaming and reinforcement learning. Marco Wiering was my mentor from the start. Due to circumstances and other causes and despite all the good input from Marco I never really started a good research.
Then in the beginning of 2007 I found out that Bas Jacobs was looking for a subject for his master thesis and because we have worked together before, we decided to start a research together. After speaking with Marco we decided to go for Othello in combination with several learning techniques. In 2005 Bas and I, together with Wouter Tinus and Tina Mioch, had already done some research on reinforcement learning and Othello, so that was a good base to start from. In April 2007 I finally started with a good research and now in January 2008 I have finished it.

I would like to take this opportunity to thank Bas Jacobs for being there at the time I really needed to start my master thesis and for the time we worked on this project together. We had great fun and did a lot of work.
Of course I would like to thank Marco Wiering for his support during all my unfinished researches and finally helping me to set up this research. And of course for all his input during this research.
Last but not least I want to thank all the people who supported me during my long thesis period and who kept me going until I was able to finish this research.

Abstract

Games like chess and Othello have been subjects of research for many years in the field of Artificial Intelligence. Because of their complexity and yet their simple rules and fully observable and deterministic nature these boardgames are very interesting subjects of research.
This thesis will focus on the game Othello. Othello is a boardgame with simple rules and a simple strategy which is easy to learn for humans. Mastering a good strategy on the other hand is difficult. Because of the many possible moves and its huge statespace it is hard to look ahead many moves.
At this moment there are several computer programs that cannot be beaten by human players anymore. They use advanced knowledge and look ahead several moves and use brute force to calculate good moves.

The subject of this research is about learning to play Othello without a priori knowledge using neuroevolution techniques and reinforcement learning.
Using a neural network as a function approximator to evaluate the boardstates, three different neuroevolution techniques are compared: SANE, ESP and NEAT. This thesis describes these techniques and shows results of experiments with learning Othello against a random Othello player. The best technique, NEAT, is also tested against some more sophisticated deterministic players. The main conclusion is that NEAT is a potential good technique to learn to play Othello.
This covers Part I of this thesis which is a collective research of Michiel Vuurboom and Bas Jacobs.

Part II contains the research of Michiel Vuurboom which uses NEAT and combines it with the reinforcement learning method TD(λ) to improve NEAT. First TD(λ) is explained and tested against the random and deterministic opponent which shows that TD(λ) is able to learn against them all except one, the Positional opponent.
Then NEAT is extended with TD(λ) resulting in NEAT-TD(λ) which is then tested against the random and the deterministic players.
Extending the sophisticated random global search of NEAT with the local search of TD(λ) should possibly result in an improvement for NEAT.
The experiments performed with NEAT-TD(λ) show that this is not the case. Possibly due to sub-optimal parameters for the techniques and other causes explained in this thesis, NEAT-TD(λ) only performs as good as NEAT in most experiments and even worse in one experiment: against the Positional opponent.
The main conclusion is that NEAT-TD(λ) might be promising, but it needs some adjustments and improvements.

Contents

Chapter 1

Introduction

"Failure is always the best way to learn,
retracing your steps 'til you know,
have no fear your wounds will heal."
Failure, Kings of Convenience

1.1 Research motivation

Game playing has always been an interesting part in the field of Artificial Intelligence. Games like chess and Othello have been subjects of research for many years. These games are interesting because of their complexity and their many possible game states while they are also fully observable and deterministic. And because of a well defined set of rules they are easy to implement.

There are many human players who play these games at an excellent level, although since a few years the computer beats man in these games. In 1997 the chess program Deep Blue defeated the world champion for the first time [Hsu04]. In 1980 an Othello program called The Moor won a game against the reigning world champion and after 1997 world champion Othello players are no match for Othello computer players [Bur97].

The current techniques to create a great computer player for chess or Othello use a priori knowledge of good strategies and use advanced search techniques to look ahead as many moves as possible. Of course a computer can look ahead much further than a human being and as computer power increases, this gap between computer and human skill level will also increase.

Although computers can beat man playing chess and Othello, that does not mean these computer players are intelligent. They are fast, they have a very large memory, but they use strategies that they did not invent themselves; they use human knowledge combined with computation power.

What if we can create a computer player that can learn playing games like Othello and chess without a priori knowledge about good strategies? Will they be able to learn to play a good competing level? Will they be able to learn the same good strategies as humans?

In 2005 we, Bas Jacob and I, made a start with research on learning to play the game Othello without a priori knowledge using reinforcement learning and neural networks resulting in a player who can defeat a novice player. Based on this result we did further research on the game Othello and this time using neuroevolution techniques.

1.2 Problem statement

Neuroevolution techniques seem promising, although at this moment there has not been that much research done on game playing such as Go and Othello using these techniques. In [Lub01] research is done on using the neuroevolution technique SANE to learn to play Go. Another example is [Per01] where the neuroevolution technique ESP is used to learn to play Go. In both articles only small boards (up to 7x7 positions) of Go have been used, because of the complexity of the game. Games like Othello and Go have huge state spaces and relatively simple rules and are therefore interesting subjects for research on neuroevolution techniques.
With a good neuroevolution technique as a start we think it must be possible to learn to play Othello at a good level without any a priori knowledge. But to be sure of that, we first need a good neuroevolution technique.
Therefore our shared research goal for this research is: *Find the neuroevolution technique that is best at learning to play Othello.*
For this we research some of the best known neuroevolution techniques and find out which techniques have potential and then compare them to find out which one is best at learning to play Othello. We will compare the techniques in terms of learning potential and learning speed.

Because in 2005 we discovered that the reinforcement learning technique TD(λ) was able to learn to play Othello, I want to find out whether a good neuroevolution technique combined with TD(λ) will perform even better. So besides our shared research and research goal that covers chapter 2, 3 and 4, I have my own research covered in chapter 5, 6, 7 and 8.

The research goal I defined (based on the outcome of our collective research): *How can the neuroevolution technique NEAT be combined with TD(λ) to perform even better than the techniques on their own?*

Our expectation is that all the techniques we compare will perform at least as good as the player created in 2005 at playing against a random opponent. One problem with the research in 2005 is that there is a bug in the software (explained in chapter 6), so the results of that research are not that good to compare to.
My personal expectations of my research on combined techniques is that they will perform slightly better than the neuroevolution techniques and the TD(λ) techniques alone.

1.3 Structure of the thesis

This thesis is split into two parts. The first part is a collective research about the comparison of the three neuroevolution techniques SANE, ESP and NEAT. In this part the game Othello is explained as well as the different strategies and the different opponents used in the experiments in *chapter 2*. The three neuroevolution techniques are explained in detail in *chapter 3* including their implementations used for the experiments. The experiments are described in *chapter 4* including all results and the first conclusion.

Part II is my own research involving the combining of the neuroevolution technique NEAT and the reinforcement technique TD(λ). It starts with *chapter 5* which explains TD(λ) and then in *chapter 6* it shows the experiments and results of learning Othello using TD(λ). How NEAT and TD(λ) are combined is explained in *chapter 7*. The experiments and their results are described in *chapter 8*.

Chapter 9 and *chapter 10* contain the conclusions and recommendations of both Part I and Part II.

1.4 Division of tasks

Part I is a collective research of both me and Bas Jacobs. All implementations of the techniques, the chapters in this thesis and the conlusions and recommendations of Part I are done together.
All research, including all implementations, all experiments, all conclusions and recommendations of Part II are my own work.

Michiel Vuurboom

Part I

Neuroevolution and Othello

Chapter 2

Othello

Othello is a derivative of the the Go family of games, and existed since the nineteenth century under the name Reversi. In 1974 the game Othello was formalized in Japan. Like Go, the game Othello is about capturing territory of your opponent. It is a two-player game on an 8x8 board with black and whites pieces. The initial board setup is shown in *figure 2.1*.

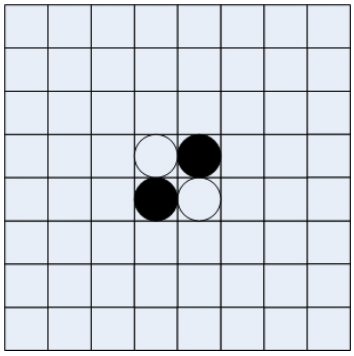

Figure 2.1: Initial boardstate

Each player takes turns placing pieces on the board. A player may only move to an open space that causes an opponent's piece or pieces to be flanked by the new piece and another one of the player's own pieces. The opponent's pieces are then captured. Pieces may be captured vertically, horizontally and diagonally. *Figure 2.2* shows the legal moves for black for the given board pattern.

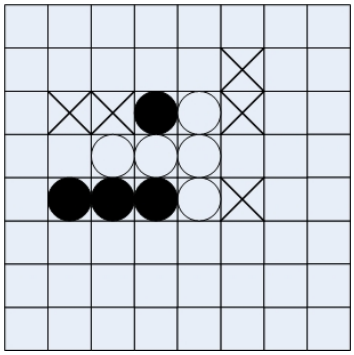

Figure 2.2: Legal moves for black

Once a move is made the captured pieces are flipped. *Figure 2.3* shows the board layout resulting from a move by black in the second row of the sixth column. The game is continued until there are no legal moves available for either player. If a player has no legal move he has to pass. The winner is the player with most pieces in the final board configuration.

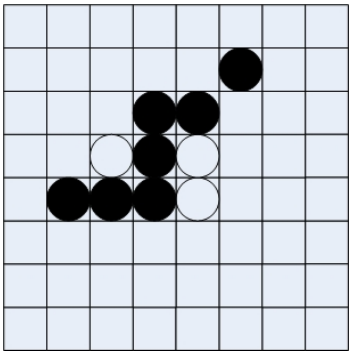

Figure 2.3: After black's turn

2.1 Playing Othello

There are different strategies for playing Othello. A player can focus on capturing the corners and edges, or he can use a strategy where capturing corners is only a sub-goal. The best human players use very sophisticated strategies and they try to look as many moves ahead as possible.

An Othello game can be divided in an opening-game, a mid-game and an end-game. Of course the end-game (approximately the last 20 moves) is the final battle in which each player tries to capture as many stones as possible. The foundation for that end-game is the mid-game. The first moves in the opening-game are important for a good mid-game. Two general classes of mid-game strategies exist in Othello: the *positional strategy* and the *mobility strategy*.

The positional strategy is simpler than the mobility strategy, but also inferior. Using a positional strategy the player has the immediate goal to capture as many stones as possible. To accomplish that, he will try to capture the edges to ring the opponent and he will try to capture the corner places at any given time because a piece in a corner can never be captured. A game with two players using a positional strategy will end up in an arms race with both players trying to get the upper hand.
The positional strategy is an easy strategy to understand and to learn and it is also easy to implement in a computer player. Most new and novice players use this strategy.

The mobility strategy on the other hand is much more complicated, but also superior. It is based on the notion of mobility: forcing the opponent to give up available moves until the player is in a position to decide exactly where the opponent will have to move. To

accomplish this the player attempts to control the centre of the board, forcing the opponent to surround the player's pieces. The opponent will be forced to surrender corners and edges in the end game because of what the player does in the mid-game. A mobility strategy can be characterized by a low piece count and a large number of available moves for the player during the mid-game. Then the opponent will have many pieces and only a few available moves.
The mobility strategy is difficult to learn. Not only for a human player is this hard to master, but it is also difficult to make a computer learn this technique [Bil90].

2.2 Computer players

Human players can only look a few moves ahead. Computer programs can compute many steps ahead, only limited by their memory capacity and speed (or the amount of time one is willing to wait for the calculations to complete). It should be stated that expert human players do not scan any more moves ahead than novice players [Gro65]. Most Othello programs use a priori knowledge for playing the game. They use an opening-book for the first few moves and use complex search algorithms in combination with different strategies to decide the next move.
By now computers are fast enough to compute many steps ahead and beat the best human players in playing Othello. But they can beat human players only because of their computing power. Search-algorithms have been developed and evolved to very fast algorithms and many strategies for playing Othello have been developed in the last 40 years [Rus95].

Current expert computers players, like WZebra [And04] and Edax [Del04], can beat all human players. They use sophisticated pattern recognition and a notion of mobility to play a mobility strategy. An opening book (a large database with opening moves and their desirability) is used for the opening-game. Then an advanced search tree is used to play the mid-game using mobility strategy. The end-game is usually played by calculating the last moves (up to the last 20 moves). By calculating the end-game, the computer already knows at the end of the mid-game whether it can win the game or not. It will then try to maximize its score, or minimize its losses using look ahead to the last move of the game.
Players like WZebra, which is one of the best computer players at the moment (and free to download and use), use advanced a priori knowledge and well defined strategies combined with brute force computation to play the game. They are optimized to play Othello with the mobility strategy. All knowledge was implemented and nothing was learned.
It would be interesting to find out whether it is possible to have a computer learn the mobility strategy. Some research has been done on this subject.
In [And02] the authors claim to have developed an Othello learning player that is capable of learning a mobility strategy using the neuroevolution technique NEAT. Although the player does not play at an expert level, it was able to learn the mobility strategy.

2.3 Othello opponents

For this research several several different opponents have been created to test against. They consist of non-deterministic and deterministic players.

2.3.1 Random opponent

The Random opponent is a very simple player. It does not use any strategy or board evaluation at all and just picks a random move from the possible legal moves.
In *chapter 6* it is explained that the Random opponent used in earlier research ([Jac05]), where this research is based upon, contained a bug that has been fixed for this research.

2.3.2 Positional opponent

The Positional opponent is a deterministic opponent. It plays using a positional strategy using a list of boardposition values to evaluate the board state.
When this player can make a move, it looks at the list of possible moves and it evaluates the resulting board states using the position values in *figure 2.4*.

100	-20	10	5	5	10	-20	100
-20	-50	-2	-2	-2	-2	-50	-20
10	-2	-1	-1	-1	-1	-2	10
5	-2	-1	-1	-1	-1	-2	5
5	-2	-1	-1	-1	-1	-2	5
10	-2	-1	-1	-1	-1	-2	10
-20	-50	-2	-2	-2	-2	-50	-20
100	-20	10	5	5	10	-20	100

Figure 2.4: Othello position values

It just accumulates all the position's values of the positions where the player has a stone. The board state with the highest evaluation is the best, so the move that will result in that state will be chosen.
As stated before, the corner positions are important, so they have high values, and the positions adjacent to the corners are bad positions to be in, because it makes it possible for your opponent to capture the corner position, so these positions have very low values.
This player does not use look ahead. It just evaluates the next moves. This is done because looking ahead takes time, and time is valuable in the many experiments in this research.
This Positional opponent is a good opponent for novice players, but can be beaten by more experienced players.
When playing against a random opponent this player wins 85% of the games. This is tested by playing 10.000 games against the random opponent and this was repeated 10 times to get a good average.

2.3.3 Mobility opponent

The mobility player uses a simple form of the mobility strategy. Because a sophisticated mobility strategy is not possible without looking more moves ahead, this player is not an advanced mobility player. It does use a notion of mobility, defined as the number of legal moves a player can make in a certain position.

This player's objective is to maximize the number of corner squares occupied by its own discs while minimizing the number of corner squares occupied by opponent's discs, and to maximize its own mobility while minimizing its opponent's mobility.
It uses the evaluation in *figure 2.5*.

$$\boxed{Eval(s) = w_1(c_{player} - c_{opponent}) + w_2(\frac{m_{player} - m_{opponent}}{m_{player} + m_{opponent}})}$$

Figure 2.5: Board evaluation function of the Mobility opponent

Where:

w_1	: a constant, 10 in this case
w_2	: a constant, 1 in this case
c_{player}	: number of corner positions occupied by player's stones
$c_{opponent}$	: number of corner positions occupied by opponent's stones
m_{player}	: the mobility of the player
$m_{opponent}$	: the mobility of the opponent

When this player can make a move, it looks at the list of possible moves and it evaluates the resulting board states by looking at the number of corner positions occupied by itself and by the opponent and looking at the numbers of possible moves of itself and of the opponent. Using the $Eval(s)$ function of *figure 2.5* all possible moves are evaluated. The board state with the highest evaluation is the best, so the move that will result in that state will be chosen.
For the sake of time, this player does not use look ahead.
This Mobility opponent is a good opponent for novice players, but can be beaten by more experienced players.
The player has been tested for 10 times 10.000 games against the random opponent and it has a win percentage of 87%.

2.3.4 TD-Greedy opponent

The TD-Greedy opponent is the final result of the research in [Jac05]. This opponent has learned to play Othello using random opponents combined with batch-learning using sample data from worldclass tournament games.
In the end it scored 83% against a random opponent and is a good opponent for novice Othello players.
This player is deterministic and it uses a neural network with 20 hidden neurons for the board evaluation. Details of this research and the player can be found in [Jac05].

Chapter 3

Neuroevolution

Creating a computer based Othello player can be done in quite a few different ways, although creating an expert one requires a priori knowledge and a lot of raw computing power. Many leaps have been made in the development of techniques capable of learning to play Othello. One of the more recent ones is the development of neuroevolution. Neuroevolution is a technique which uses genetic algorithms to train artificial neural networks.

3.1 Neural networks

An Artificial Neural Network, also known as Neural Network, is a processing unit based upon the principles of biological information processing performed by the brain [Ste96]. Key components are the neurons and the connections between them. These neurons are linked in a specific manner depending on the task a neural network is assigned to. A neural network is designed to learn by example and through examples its connections are updated in order to generate better solutions to the problem presented. A typical neural network can be seen in *figure 3.1*. The Input Layer is where information is fed to the neural network; the Output Layer gives the outcome of the neural network based upon the inputs given; The Hidden Layer allows for more complicated tasks to be learned. More hidden units and layers allow for more complicated tasks to be learned at the expense of computation time.

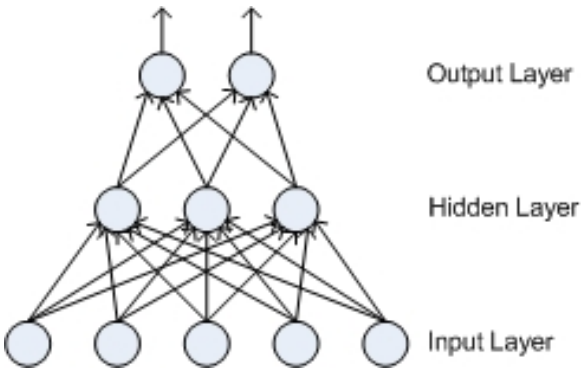

Figure 3.1: Simple feedforward neural network

Neural networks have been used in many fields such as sales forecasting, industrial process control, customer research, data validation, risk management, target marketing and more recently the gaming industry. Neural networks allow a system to map a domain state to a desired action and they are capable of generalizing over states easily which is most welcome in areas where a huge amount of states are possible.
Some of the advantages and disadvantages of neural networks as described in [Vel99] are:

Advantages:

1. Neural networks are able to learn any complex non-linear mapping / approximate any continuous function.
2. As non-parametric methods, neural networks do not make a priori assumptions about the distribution of the data / input-output mapping function.
3. Neural networks are very flexible with respect to incomplete, missing and noisy data / neural networks are fault tolerant.
4. Neural network models can be easily updated / are suitable for dynamic environments.
5. Neural networks overcome some limitations of other statistical methods, while generalizing them.
6. Hidden nodes, in feed-forward supervised neural network models can be regarded as latent / unobservable variables.
7. Neural networks can be implemented in parallel hardware, increasing their accuracy and learning speed.
8. Neural networks performance can be highly automated, minimizing human involvement.
9. Neural networks are specially suited to tackle problems in non-conservative domains.

Disadvantages:

1. Neural networks lack theoretical background concerning explanatory capabilities / neural networks as black boxes.
2. The selection of the Network topology and its parameters lacks theoretical background / It is still a trial and error matter.
3. Neural networks learning process can be very time consuming.
4. Neural networks can overfit the training data, becoming useless in terms of generalization.
5. There is no explicit set of rules to select a suitable neural network paradigm / learning algorithm.
6. Neural networks are too dependent on the quality / amount of data available.
7. Neural networks can get stuck in local minima / narrow valleys during the training process.
8. Neural network techniques are still rapidly evolving and they are not reliable / robust enough yet.
9. Neural networks lack classical statistical properties. Confidence intervals and hypothesis testing are not available.

For the disadvantages 2, 3, 7 a solution was found in the development of neuroevolution techniques.

3.2 Genetic algorithms

Genetic Algorithms (GA's), also known as Evolutionary Computing, is a population-based stochastic search algorithm based on the mechanics of natural evolution. GA's are a subset of Evolutionary Computing and are used to find approximate or exact solutions to optimization and search problems. Such applications are commonly presented by research fields such as biogenetics, physics, computer science, economics, engineering, chemistry and mathematics. For this research GA's will be used to find solutions for playing Othello.
GA's are based on, but not limited to the characteristics of natural evolutionary systems. GA evolution has 5 distinct characteristics:

1. Structures, which are a genetic representation of the solution domain
2. Structures are combined to form new, better solutions
3. Structures compete for a limited resource
4. Fitness function to evaluate a solution
5. Relative production success depends on the environment

Structures are complete individuals and can act in a given environment in order to determine their ability to execute the given task. The value given for this ability is called the fitness. Fitness is a single number given to each structure as a performance measure. The fitness is determined by doing 1 trial per structure in the environment if both the structure and the environment are deterministic, or multiple trials if randomness is involved.
A structure consists of genes to describe its characteristics. These genes combined are called chromosome or genotype. The structure of these chromosomes is manually designed. *Figure 3.2* shows a representation of a simple binary chromosome. Chromosomes can be merged to create offspring which has characteristics of 2 parents (or more). This way important characteristics of successful parents can be passed on to offspring to create better solutions. In addition to merging the genes of parents, mutation is also used to maintain genetic diversity.

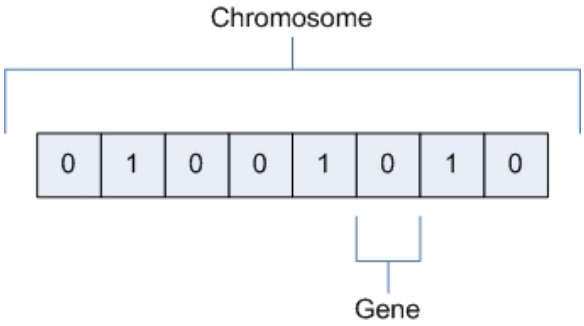

Figure 3.2: Representation of a binary chromosome

As a genotype, it is in most cases not possible to determine the fitness of the structure. Just like in nature, a genome itself is mere data, but with the data a creature can be created.

This is called a phenotype. The phenotype is used to determine the fitness of the genotype. According to this fitness a sorted list is created and individuals are selected for reproduction. Parents can be selected any way one chooses, although some selection techniques are more popular than others. Examples are Fitness Proportionate Selection, Tournament Selection and Ranked Based Selection.
Figure 3.3 shows a general genetic algorithm flow. First an initial population(*P(0)*) is generated; often with random values but predetermined topology. Secondly *P(0)* is evaluated to be able to select parents. Next the while loop is started to compute new generations. In this loop the parents(*P'(t)*) are selected for recombination which produce offspring(*P"(t)*). Now we have too many individuals, and so in this example, *P'(t)* ∪ *P"(t)* need to be sorted on fitness after which the best are kept as a new generation *P(t+1)*.

```
P(0)          ←  Generate initial population()
P(0)          ←  Evaluate population(P(0))
t             ←  0
While             Not-Terminated P(t)
do
  P'(t)       ←  Select mates(P(t))
  P"(t)       ←  Generate offspring(P'(t))
  P"(t)       ←  Evaluate population(P"(t))
  P(t+1)      ←  Select fittest(P"(t) ∪ P'(t))
  t           ←  t + 1
return P(t)
```

Figure 3.3: Genetic Algorithm Pseudo-Code

3.3 Neuroevolution

Neuroevolution is a technique where GA's are used to improve neural networks. There are many neuroevolution techniques, which can be classified in techniques which evolve the neural network weights versus techniques which evolve both the weights and the topology of the neural network. GA's which evolve both the neural networks weights and topology are also called TWEANNs (Topology & Weight Evolving Artificial Neural Networks).
When GA's are used to evolve neural networks, the network (which is a phenotype) has to be converted to a genotype to be able to reproduce. Weight values can be stored in a chromosome in different ways; *direct encoding* and *indirect encoding*. Direct encoding means having floating point values in the chromosome representing all weights. Indirect encoding can be determined by the developer.
Several neuroevolution techniques exist and for this research SANE, ESP and NEAT are compared.

3.3.1 SANE

Symbiotic, Adaptive Neuroevolution (SANE) [Mor96], [Mor97], is a reinforcement learning method which evolves a population of neurons through genetic algorithms to form a neural

network. Evolving a population of neurons instead of full neural networks makes it possible to develop partial solutions to the posed problem. *Figure 3.4* shows the basic steps for computing one generation in SANE.
The goal of SANE is to have each individual develop a solution which can be combined with others to form a complete and effective solution to the problem. Because individuals alone can not make an effective solution, symbiotic relations must be maintained with other individuals. When creating a full neural network, neurons are chosen from the population pool and combined to form a complete neural network. *Figure 3.5* shows the conversion between a genotype and its phenotype.

1. Clear all fitness values from each neuron.
2. Select ζ neurons randomly from the population.
3. Create a neural network from the selected neurons.
4. Evaluate the network in the given task.
5. Add the network's score to each selected neuron's fitness variable.
6. Repeat steps 2-5 a sufficient number of times.
7. Get each neuron's average fitness score by dividing its total fitness value by the number of networks in which it was implemented.
8. Perform crossover operations on the population based on the average fitness value of each neuron.

Figure 3.4: One generation in SANE

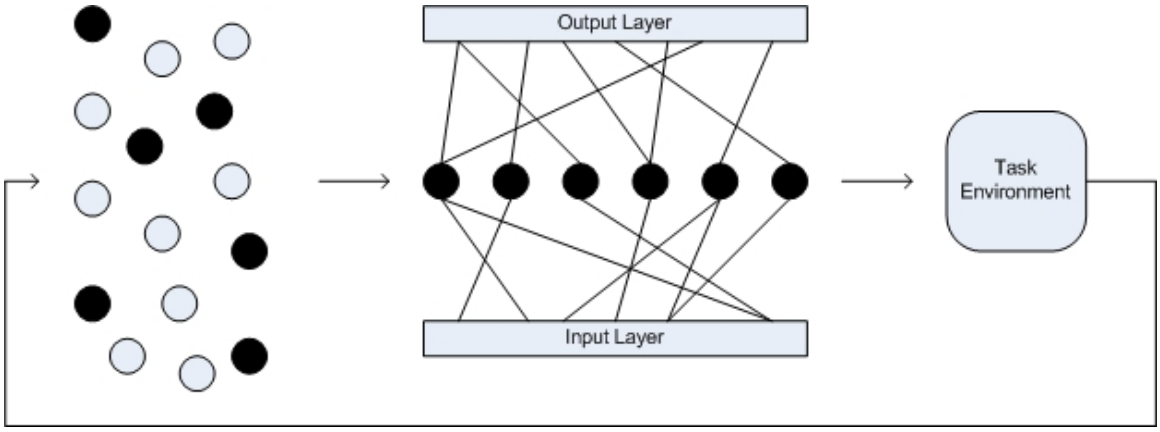

Figure 3.5: SANE, Genotype to Phenotype

Fitness can easily be determined for all individual neurons after having determined the fitness for the formed neural network. When an individual participates in a neural network, the fitness of the neural network is equally assigned to all participating neurons. This way an individual can take part in any number of neural networks. In theory all neurons should participate in neural networks with all other neurons to get an optimal weighed fitness for each neuron. Obviously this is not feasible and fitness will remain an approximation.
Specialization is an important aspect of SANE and is possible due to the individual neurons being evolved. Instead of solving the entire problem, individual neurons aim to solve a

particular aspect of the problem. Specialization is enforced knowing individual neurons cannot form a complete solution and the fitness of the neurons is based on the effectiveness of symbiotic relations it has with other neurons. Specialization prevents converging to suboptimal solutions because of the diversity in the population.

3.3.2 ESP

Enforced Sub-Populations (ESP) [Gom99], is a reinforcement learning method very similar to SANE. ESP evolves a population of neurons through genetic algorithms to form a neural network. Like SANE, ESP evolves a population of neurons, but unlike SANE, the specializations are not kept in a single parent pool. A drawback with SANE is the interbreeding of different specializations which result in a lot of individuals with similar characteristics as well as very few to no protection of new (still weak) species. ESP enforces a subpopulation for each hidden neuron of the neural network as can be seen in *figure 3.6*. Neurons in a subpopulation can only recombine with neurons from its own subpopulation. These enforced subpopulations allow a much faster specialization than is the case with SANE (where all specializations have to emerge from one large pool). Having subpopulations protects weaker species from dominant ones taking over the population. Also having neurons being placed at the same location in the neural network, and being linked to the same neurons increases learning speed and allows better learning for recurrent networks.

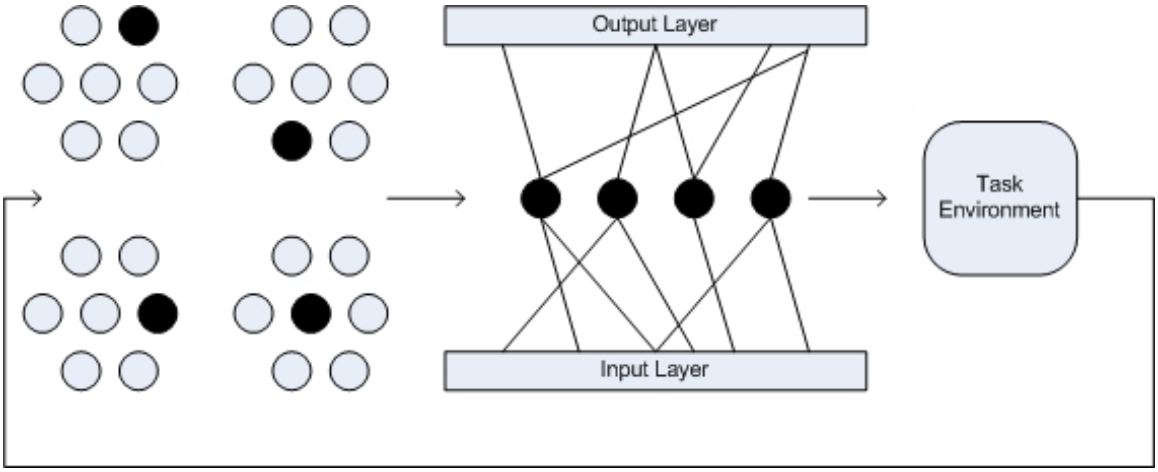

Figure 3.6: ESP, Genotype to Phenotype

3.3.3 NEAT

Neuro-Evolution of Augmenting Topologies (NEAT) [Sta02], is a reinforcement learning method, although not like SANE or ESP. NEAT is a TWEANN, a GA which evolves both weights and topology of neural networks. Like SANE and ESP weights are evolved through generations allowing a better solution to be reached. In addition, changes can be made to the topology in terms of links and nodes. This allows NEAT not only to search the search-space but also to minimize it during evolution.

The initial topology in NEAT can be setup by humans to fit the problem to be solved. Initial topologies more closely matching the optimal topology are recommended as it reduces the time required to evolve to the optimal topology. NEAT's topologies often become increasingly more complex as they become more optimal, strengthening the analogy between GA's and natural evolution.
NEAT uses direct encoding to describe network structures as indirect encoding would limit the topological search to the class of structures which would be designed for the indirect encoding. As NEAT evolves topologies, this is clearly an unwanted characteristic. *Figure 3.7* shows the genotype and phenotype of NEAT. The genotype consists of two types; node genes and connect genes. Node genes come in three types; sensor, hidden and output. Hidden nodes are removed or added through evolution. Connect genes represent the links/weights between nodes. A connect gene defines one link/weight between two specified nodes and can be enabled or disabled through mutation and crossover operators.
Specialization is also allowed by NEAT due to the historical markings assigned to each individual in the population.

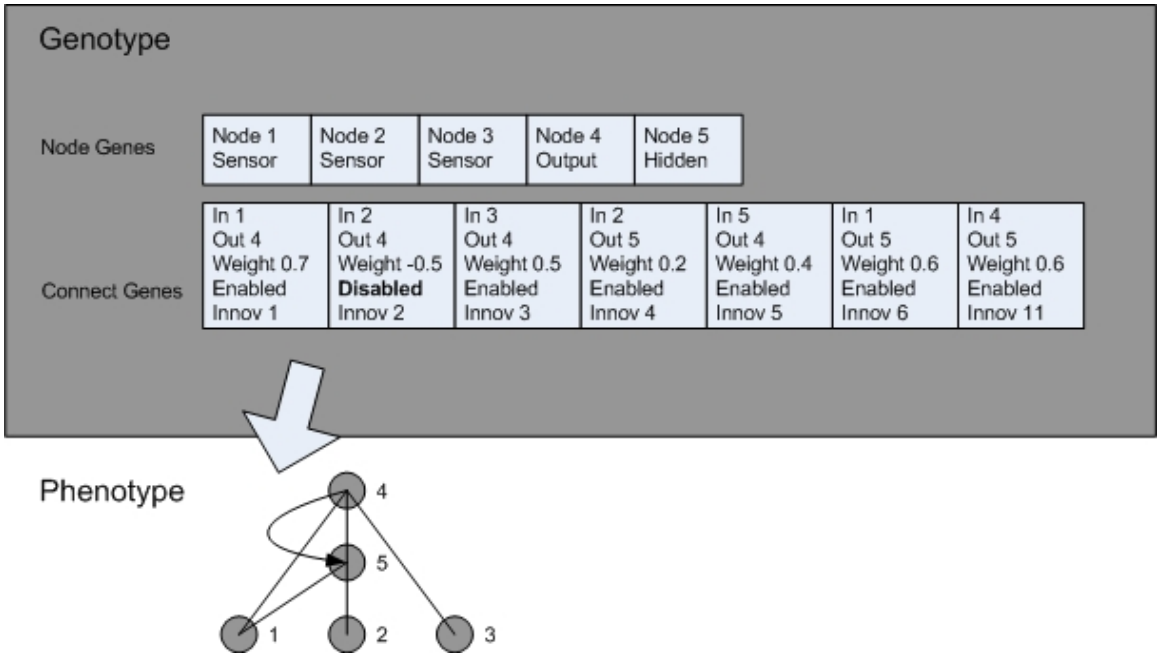

Figure 3.7: NEAT, Genotype to Phenotype

3.3.4 Other Neuroevolution Techniques

Besides SANE [Mor96], [Mor97], ESP [Gom99] and NEAT [Sta02] other neuroevolution techniques have been developed like TEAM [Ald02] and CoSyNe [Gom06]. The Eugenic Algorithm with Modeling (TEAM) is an extension of the evolution technique The Eugenic Algorithm (EuA [Pri98]). TEAM is a technique for evolving a population not only by standard crossover and mutation, but also by directing evolution. This is done by maintaining historical information on correlations between allele and fitness. The available software

contains code for evolving binary genes only. TEAM was not part of the experiments due to time constraints and remains an interesting technique to experiment with for creating Othello players.
Cooperative Synapse Neuro Evolution (CoSyNe) [Gom06] is another neuroevolution technique. CoSyNe searches at the level of individual network weights instead of neurons. Like ESP and NEAT, CoSyNe has n subpopulations, equal to the amount of network weights to be evolved. All subpopulations have an equal predetermined size. Phenotypes are created by selecting one individual from each subpopulation and inserting it at its position. Each subpopulation has individuals specifically for one position in the neural network. As CoSyNe was released after the choice of techniques, it was left out.

3.3.5 Neuroevolution and Games

There has not been much research in learning games like Othello with neuroevolution techniques. The three techniques discussed here have been used in control tasks like pole-balancing problems and similar tasks, but not in playing games.

As stated before NEAT has been used to learn a mobility strategy in Othello ([And02]). Although not like an expert player, it was able to learn mobility. The game Go has been studied using SANE and ESP in [Lub01] and in [Per01], but only for small Go boards (up to 7x7 positions). These examples show that there is still a lot of interesting research that can be done on neuroevolution and board games.

Chapter 4

Experiments

4.1 Implementation

To test the three neuroevolution techniques a Java implementation of Othello was used, along with Java implementations of the three techniques.
Java was used because of previous work on Othello [Jac05] was also done in Java, so an Othello framework ready for experiments already existed.
The three techniques were originally written in C or C++ but they all have a Java implementation as well.

4.1.1 Othello

The implementation has been divided into two main groups, Environment classes and Player classes. Environment classes contain code for the game itself, while Player classes contain (any form of) intelligence for playing a game of Othello using the Environment classes.
An abstract view of the environment implemented in Java is given in *figure 4.1*.

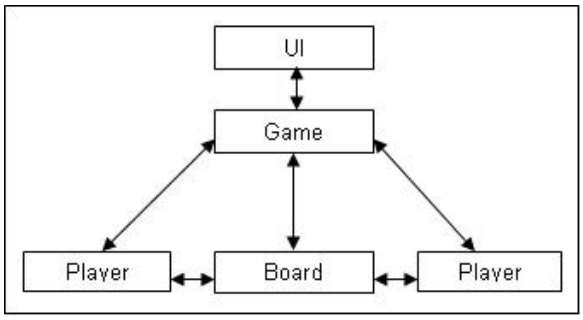

Figure 4.1: Othello Java classes

The environment has several features so it can be used for experiments, like keeping track of the scores, playing multiple games and several others.
The player classes are a collection of all implemented players and the Player interface. Each game consists of two players chosen from the available implemented players.
A collection of players used in previous work ([Jac05]) can be used as opponents for the neuroevolution techniques. There is a human player, which is in fact a user interface so a real human player can play Othello. There is also a random player who plays random moves. Also used are the Positional player and the Mobility player as explained in *chapter 2*. The last player used in this research is the player using the neural network that was learned in this previous research, the Temporal Difference player (TD player). It plays a good game

against novice players and can beat random players 83% of the time without looking ahead. During this research look-ahead was implemented for this TD-player, using first a min-max algorithm, and later on an alpha-beta search algorithm. When the TD-player looks ahead 3 moves or more it beats a random player 99% of the time.
Because the Othello environment is separated from the players, it is easy to add all kinds of different players. With this approach it was not difficult to add neuroevolution players using the code from other authors.
The neural networks used in the different techniques always consist of 64 input neurons and 1 output neuron. The number of hidden neurons and the way the neurons are connected varies.
The 64 input neurons represent the 64 board positions. The input is 1 if a board position contains a piece of the player, -1 if the position contains a piece of the opponent and 0 if the position is empty.
The output neuron is a sigmoid activation function and represents the evaluation value of the given board state.
The fitness function used in the genetic algorithms is a win percentage after playing 50 games. At the end of each generation, the best neural network is allowed to play 1000 games to set the fitness off the champion of that generation.

4.1.2 SANE

The JavaSANE package contains the source code for the Hierarchical SANE system, based on SANE-C by Moriarty, [Mor96], but rewritten in Java. This package is designed to be an easy starting point for applying JavaSANE to a new domain.
For this research it was just a matter of rewriting the fitness function so that it plays Othello to evaluate the neural network. Of course some parameters had to be tuned so the correct topology for the neural network was used.
Several features have been added to this package to make it easier to analyze the results and to monitor evolutionary progress.

4.1.3 ESP

The Java ESP package contains the source code for the Enforced Sup-Populations system which is nearly a direct port of the ESP C++ package that was used for research in [Gom99]. It supports different kinds of neural networks like Simple Recurrent Networks, Second Order Recurrent Networks and Fully Recurrent Networks, but for this research only simple feed forward networks are used.
With Java ESP it was also not difficult to add an Othello configuration and fitness function and integrate it in the Othello environment.

4.1.4 NEAT (JNeat and Anji)

In this research two Java implementation of NEAT have been tried, JNeat and Anji.
First JNeat was used for this research. JNEAT was written by Ugo Vierucci based on the original C++ package by Kenneth Stanley which was used in the NEAT research [Sta02].

JNeat is an extensive package with a user interface to monitor progress and it uses a complex object model. It was not that hard to integrate it with the Othello environment and create the right fitness function, but it was difficult to add and change the code to make it easier for this research to monitor progress and analyze results. In fact this package was not really well-written and the Italian comments did not make things easier.
On top of all this it did not turn out to work that good. It never really learned anything despite all different approaches and configurations.
So JNeat was put aside and Anji was used instead. Anji did learn and was easy to implement. Anji was written by Derek James and Philip Tucker based on descriptions of NEAT in papers published by Kenneth Stanley and Risto Miikkulainen ([Sta02]). It was not directly based on the NEAT C++ package, so it probably differs in some aspects.
Anji is very well written and is easy to configure and adjust to make it ready for the Othello experiments.

4.2 Initial experiments

Before running the final large experiments several initial experiments have been performed to fine tune the different techniques. All three techniques have several parameters to tune like selection mechanisms for the genetic algorithm, the different mutation rates and several others. Also the knowledge representation of the Othello player, the neural network, had to be tuned to see what (initial) topology would work best.
To compare three different techniques, several parameters were set the same for all techniques, so it was possible to compare the outcome. These parameters include population size and number of games each individual can play. This way, the three techniques have the same number of evaluations.
Each experiment was allowed 500.000 evaluations. An evaluation is one game of Othello against a random playing opponent. A random opponent was selected as most skilled players require a lot of time per game. Also when playing against a skilled player, which is often deterministic, the evolutionary technique only learns a limited amount of states. Every generation consists of 100 individual neural networks and they all play 50 games of Othello each generation. This means each experiment took 100 generations. Each experiment was repeated 3 times. This should be enough to tune the parameters.

4.2.1 Tuning SANE

To tune SANE several experiments have been performed. The main focus for tuning SANE was finding out what neural network works best for SANE. To test this a fully connected network was tested with different numbers of hidden neurons to find out what network size would work best. Also different mutation rates were tested.

Testing network sizes: 20, 40 and 60 hidden neurons.
These networks have been tested with fully connected neurons with the default SANE parameter for the mutation rate which is 20%. Although the final results did not differ that much, except for the obvious difference in learning speed, it seemed that 40 hidden neurons had a better result in the long run. The number of hidden neurons, 40, may look a bit

arbitrary, but it was not only chosen because of the outcome of this experiment, but also because of past experiences with Othello and neural networks and the expectations that 20 would be too few to learn and 60 too many to learn fast enough. The fact that the results showed not much difference between the network size, justifies an arbitrary choice.

Testing mutation rate: 20%, 40% and 60%
The mutation rate is the chance that the weight of a neuron is mutated every generation. The default SANE value is 20%. This value was tested against 40% and 60% with a fully connected network. The results of these experiments did not differ that much. It looks like 40% performs slightly better than the other two in the long run. At least it has a better average fitness in the top 10 in every generation in the end. The overall average fitness of the population is about the same in every generation for the different mutation rates.
Several other parameters were not tuned because they did not seem to matter that much or appeared fine with pre-initial experiments.

4.2.2 Tuning ESP

For ESP both the network size and the mutation rate have been tested. The first goal was to find out what size neural network works best with Othello. The second goal was to find out which mutation rate offers a good learning rate.

Testing network sizes: 20, 40 and 60 hidden neurons.
Fully connected networks were tested with default ESP parameter for mutation rate which is 40%. After 100 generations it was clear that a network with 20 hidden neurons does not perform as good as the ones with 40 and 60 hidden neurons. The ones with 40 and 60 hidden neurons do not differ that much. Both show a learning curve and have the same fitness in the end.
Because of SANE and other past experiments with Othello the 40 hidden neurons seems a good network size.

Testing mutation rate: 20%, 40% and 60%
With a fully connected neural network with 40 hidden neurons the different mutation rates were tested. After 100 generations it was clear that a mutation rate of 40% is the best. It has a fast learning rate and is still learning at the end of the experiment. Both 20% and 60% have a lower fitness in the end.
Several other parameter were not tuned because they did not seem to matter that much or appeared fine with pre-initial experiments.

4.2.3 Tuning NEAT

Tuning NEAT (with the Anji implementation in this research) took some more time. Pre-initial experiments made it clear that several parameters had to be tuned. For NEAT the parameters *weight mutation rate*, *connection mutation rate*, *neuron mutation rate* and the *speciation threshold* were tested.
The initial topology at the start of each experiment is a fully connected neural network with 10 hidden neurons.

Testing weight mutation rate: 0.005, 0.01, 0.1, 0.25, 0.5
The weight mutation rate is the probability of existing connection weights being mutated. Very low values were tested here. Because pre-initial tests showed that a low value for this was good, several tests had to be performed to see how low this value should be. In the end 0.01 turned out to be a good value.

Testing add connection mutation rate: 0.03, 0.06, 0.2, 0.3, 0.5
The *add connection mutation rate* is the probability of new connections being added with an initial weight with a random value from a normal distribution.
First the values 0.03, 0.06 and 0.2 were tested, but it turned out that 0.2 performed much better than 0.03 and 0.06. So apparently a higher value was needed. Therefore 0.2 was tested against even higher values, 0.3 and 0.5. The results of these tests did not differ that much. It seemed that 0.2 performed just a little better than the other two. At least it is still learning at a higher rate than the other two after 100 generations.

Testing add neuron mutation rate: 0.1, 0.02, 0.005, 0.001, 0.0005
The *add neuron mutation rate* is the probability of new nodes being added to an existing node in the neural network.
After the first tests with 0.1, 0.02 and 0.005 it turned out that 0.005 performed better than the other two, so 0.005 was tested against 0.001 and 0.0005. The mutation rate of 0.005 turned out to be the best because the other two values did not seem to learn anymore after 100 generations.

Testing speciation threshold: 0.1, 0.2, 0.4, 0.5, 0.6
The speciation threshold is the compatibility threshold used to determine whether two individuals belong to the same species.
After the first tests with values 0.1, 0.2 and 0.4 showed that 0.4 was the best value another test was done with 0.4 against 0.5 and 0.6. The results are close, so it is probably not necessary to do more test. The value 0.6 turned out to be the best value for the speciation threshold.

4.3 Final experiments

Final experiments were conducted once the tuning of the different neuroevolution algorithms was completed. Tuning was needed to assure maximum performance of all three techniques when comparing them in the Othello environment. The final experiments were done allowing a lot more evaluations per experiment than was the case with the initial experiments. Obviously this was done due to time constraints and shorter tuning experiments did set a trend to allow proper variable tuning. Each experiment was allowed 2.000.000 evaluations, and each experiment was repeated 10 times. One evaluation means one game of Othello. Each fitness measure counts as one evaluation and for each generation every individual was measured, even if the individual was already measured. Because individuals were trained against an opponent which makes random moves, the extra fitness measures means the assigned fitness will be more accurate.
It should be noted that the size of neural networks and thus the computation time was not taken into account.

4.3.1 Time

Running experiments takes time. All experiments are repeated 10 times and each individual experiment of each technique against the random opponent takes about 8 hours to complete. This means that repeating this 10 times will take 80 hours for each experiment against the random players.
The experiments against the deterministic opponent, as described in the last paragraphs of this chapter, take even more time, up to 10 hours for each individual experiment.

Because of the amount of time it takes to perform an experiment choices had to be made for which experiments are done and which are left out. Probably more and better results were possible if there was more time available. More on this in the last chapters.

4.4 SANE - Final experiments

4.4.1 Properties

As said before SANE was trained up to 2.000.000 evaluations per experiment. From the initial experiments the following parameters were found to be most optimal for learning Othello against a random moving opponent:

- Mutation rate = 0.4
- Neural network = Fully connected feed forward
- Networks created per generation = 100
- Number of hidden neurons = 40

4.4.2 Running Experiments

After running each SANE experiment ten times, it was clear SANE was performing on par with standard reinforcement learning techniques such as Temporal Difference learning; scoring 83% [Jac05].

As a performance measure for every generation the champion of the generation was allowed to play 1000 games in order to get a more accurate fitness. This can be seen in *figure 4.2*.

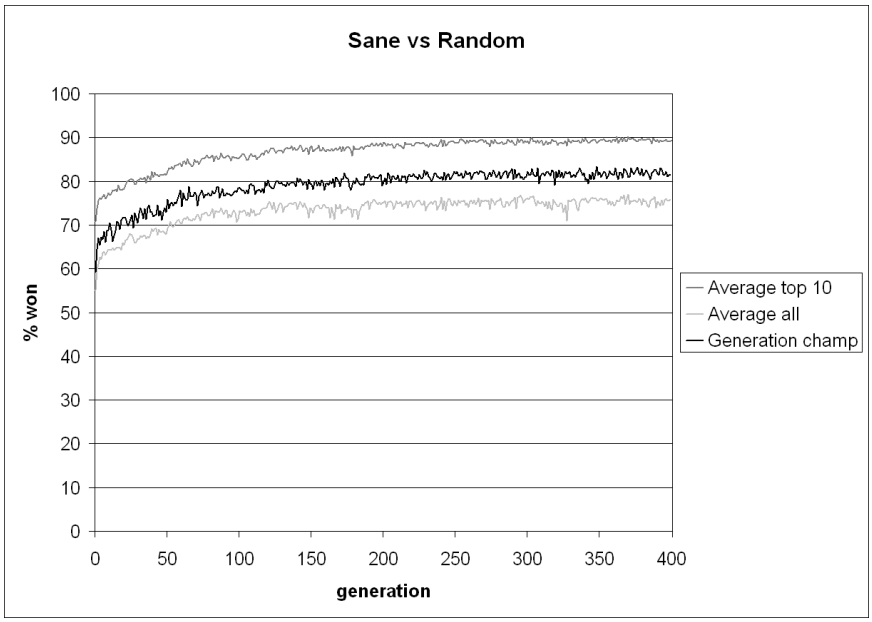

Figure 4.2: SANE, experimental results - 2000 neurons

SANE certainly learns properly, but not more than any default reinforcement learning technique (83%). Sane also ends up with a 82% - 83% score.
When comparing the *champion* to the win percentage of all the evaluated networks (*average all* in *figure 4.2*), it is clear the champion performs better than all the evaluated networks. Of course this was to be expected but seeing a gap of 6% wins suggests a large spread in fitness. Unfortunately this is a characteristic of SANE having all neurons in one large pool and thus allowing very bad networks to be formed as well. This might also be the reason for all of the ten experiments to have a rather large spread in win percentage of the champion at the end of each experiment (win percentages between 77% and 86%). The initial populationsize (2000) might also have been too large, resulting in new neurons being selected every trial and thus preventing proper learning.
There also is a difference of 8% - 9% between the *champion* percentage and the *average top 10* percentage. This is because the top 10 consists of the individuals with the 10 highest fitness scores and this fitness is determined after playing only 50 games. The best individual plays 1000 games and its fitness will be the champion fitness. This champion fitness is bound to be lower than the average top 10 fitness because of the inaccuracy of playing only 50 games compared to playing 1000 games.

Two more settings have been tested; 400 neurons (*Figure 4.3*) and 800 neurons (*Figure 4.4*) as populationsize to have each neuron participate in a network 10 times and 5 times on average respectively. As can be seen both population sizes of 400 and 800 performed slightly better than the initial populationsize of 2000. Unfortunately this is no real improvement to make SANE perform significantly better than it did with a population of 2000. Only the

experiment with 400 neurons results in a score higher (about 85%) than the original which suggests that maybe the neuron pool can be made even smaller for better results, although a score of 85% might be the top for SANE. No more experiments using SANE have been performed due to time constraints.
Although mutationrate might be high (40%), tuning tests showed this to not influence the evolution significantly.
Alternative selection and replacement methods could improve SANE's performance.

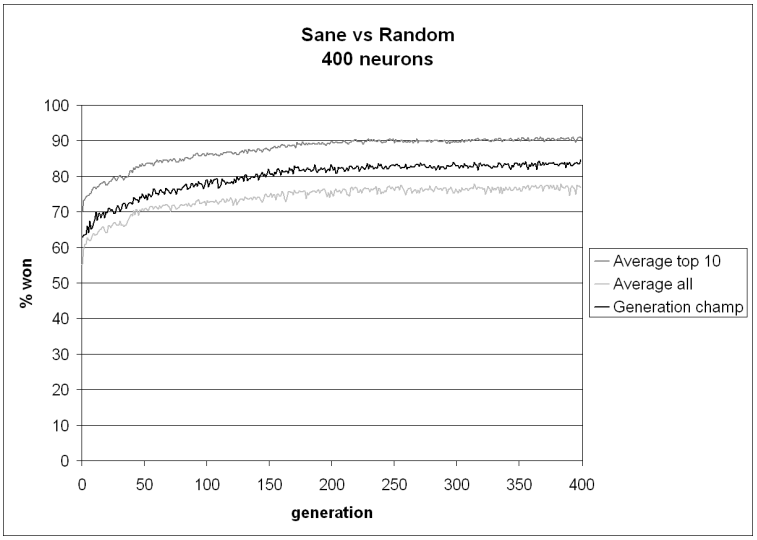

Figure 4.3: SANE, experimental results - 400 neurons

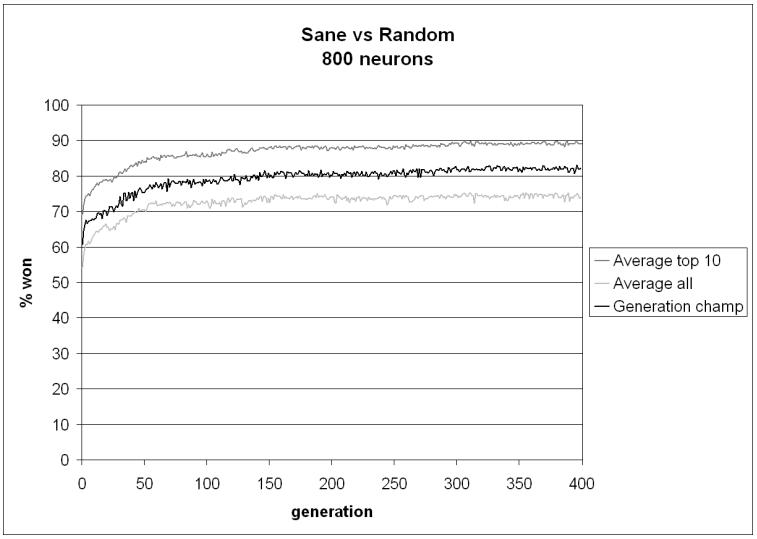

Figure 4.4: SANE, experimental results - 800 neurons

4.5 ESP - Final experiments

4.5.1 Properties

ESP was also trained up to 2.000.000 evaluations per experiment. The following parameters were considered most effective for ESP:

- Mutation rate = 0.4
- Delta coding = true & false
- Networks created per generation = 100
- Neural network = Fully connected feed forward
- Number of games per network = 50
- Number of hidden neurons = 40

4.5.2 Running Experiments

For ESP two different settings were used; delta coding and no delta coding. At first ten experiments were performed with delta coding enabled. Delta coding allows ESP to create more diversity when no new champion had been discovered for a while.

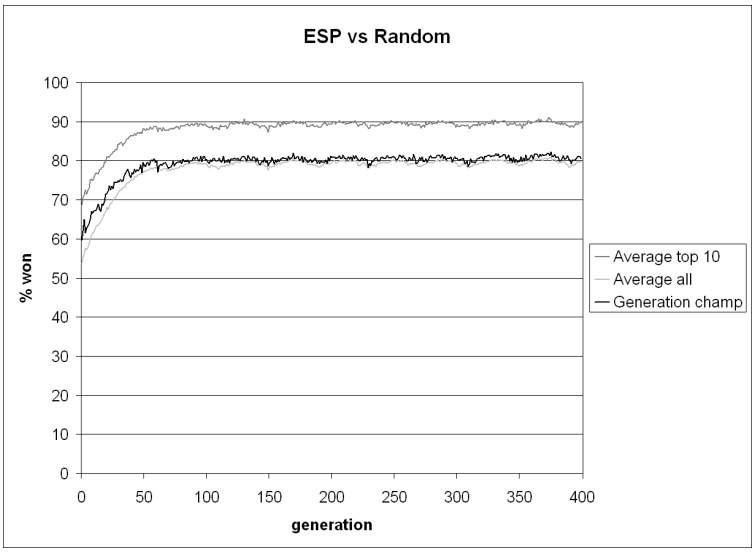

Figure 4.5: ESP with delta coding, experimental results

ESP with delta coding performs almost as good as the earlier experiments done with SANE. The first experiment was done using a subpopulation size of 100. *Figure 4.5* shows the results for ESP with delta coding. The champ in ESP reached 80%. The humps in the graph are times when delta coding is done. Unfortunately this prevented ESP from getting a score above 82%. More tests with setting the delta coding parameters might result in better performance. ESP with delta coding reached its maximum at generation 100.
Before a set of delta coding parameters was tried, delta coding was disabled. This resulted in much better performance which can be seen in *figure 4.6*. With delta coding disabled

ESP was allowed to continue evolving without the continuous setbacks resulting from delta coding. Without delta coding ESP learns much more smoothly and reaches the maximum obtained by ESP with delta coding at generation 70. Now this is nothing new, but without delta coding ESP continues evolving and has converged at generation 200 having a winning percentage of 87%.

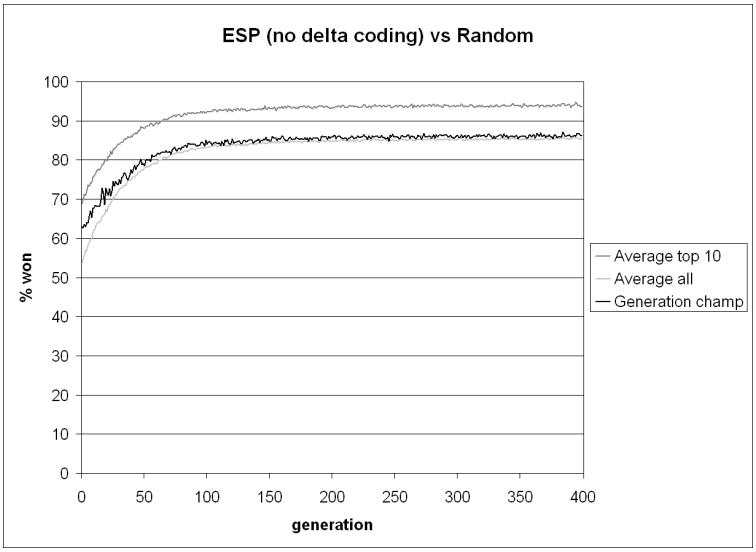

Figure 4.6: ESP without delta coding, experimental results

These first two experiments are performed with a subpopulation size of 100 and with 40 subpopulations (one for each hidden neuron). Which means that when creating a network in the population each neuron will be selected only once each generation.
In [Gom99] the neurons in de the subpopulations are tested in different neural networks for a good evaluation of the neuron. That is why two extra experiments have been done with smaller subpopulation sizes. A subpopulation size of 10 and 20 were tested, so each neuron is evaluated 10 and 5 times respectively. No delta coding is used in these experiments.
The results are shown in *figure 4.7* and *figure 4.8*.

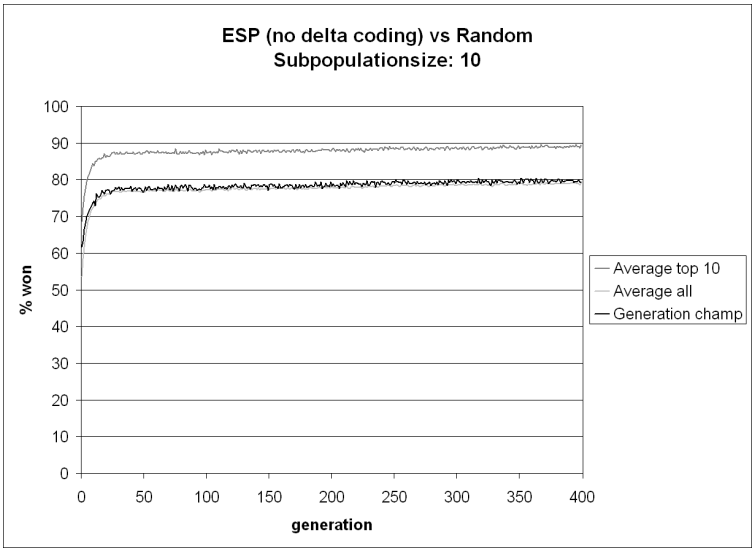

Figure 4.7: ESP subpopulation size of 10 without delta coding

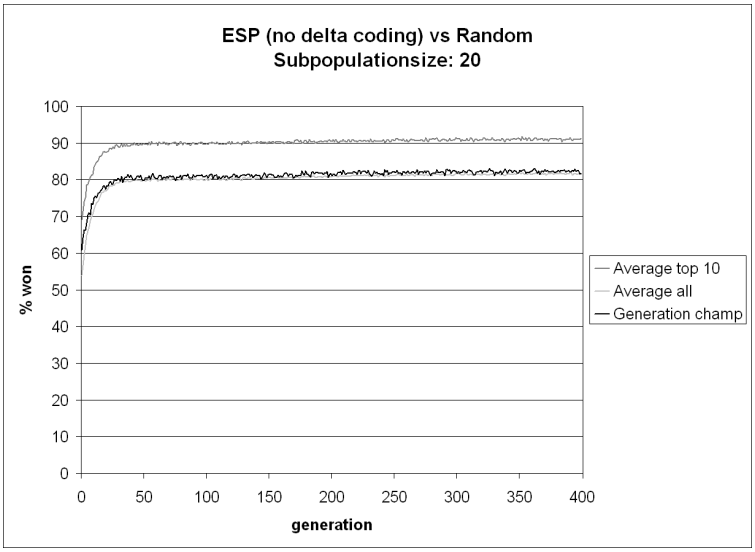

Figure 4.8: ESP subpopulation size of 20 without delta coding

The results show that when using smaller subpopulation sizes and thus evaluating a neuron more than once each generation, does not give better results. The endresults of the experiment with the subpopulation size of 100 are better.
However the learning speed is significantly higher in these last experiments. A maximum is reached around generation 50 instead of generation 150 with the larger subpopulation size. Perhaps there is some potential using smaller subpopulations which can be exploited by using delta coding to create more diversity after a stagnation of the learning speeds.
So several new experiments have been performed using the two different subpopulation size 10 and 20 and using two different stagnation values 40 and 100. This stagnation value is the number of generations in the experiments without improvements. So a stagnation value of 40 means that when there has not been a significant improvement over the last 40 generation, delta coding is used on the current population.
The results of these four new experiments are shown in *figure 4.9*, *figure 4.10*, *figure 4.11* and *figure 4.12*.

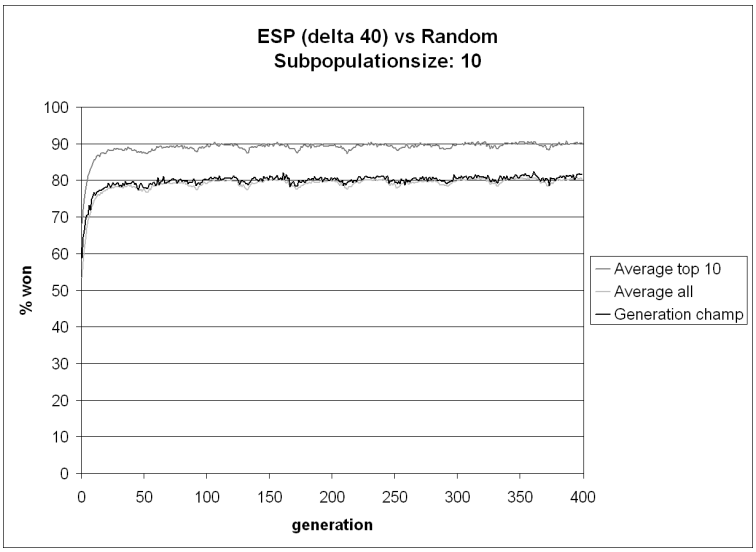

Figure 4.9: ESP, subpopulation size of 10, delta coding stagnation value 40

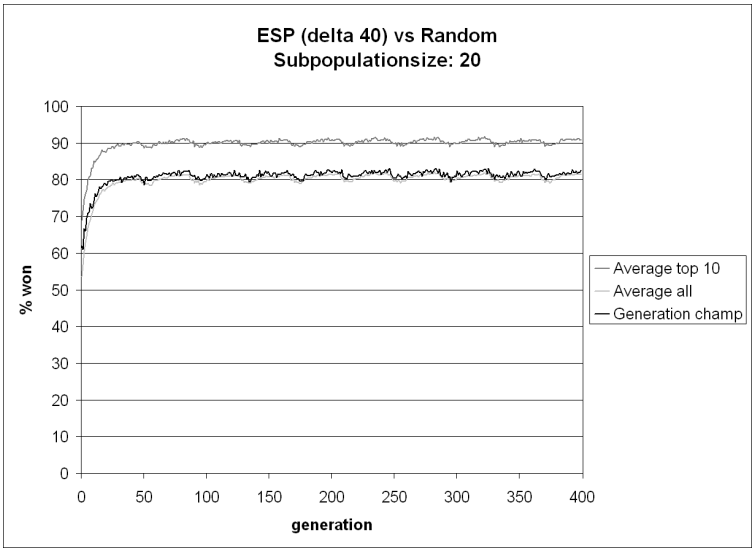

Figure 4.10: ESP, subpopulation size of 20, delta coding stagnation value 40

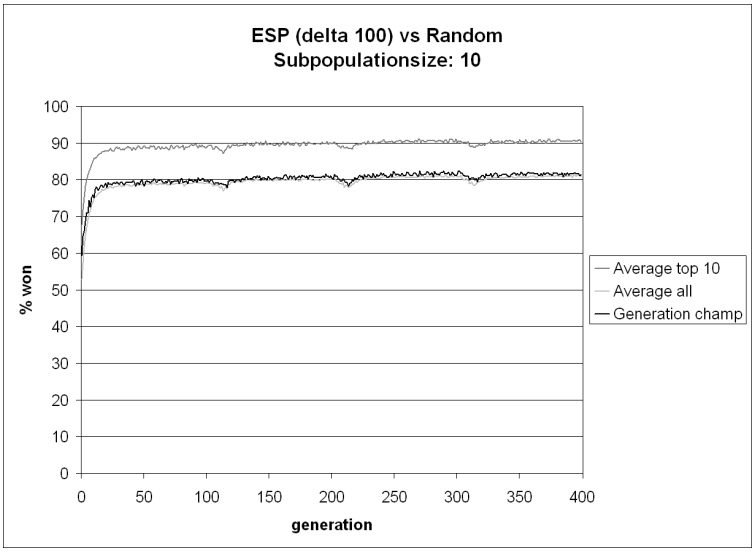

Figure 4.11: ESP, subpopulation size of 10, delta coding stagnation value 100

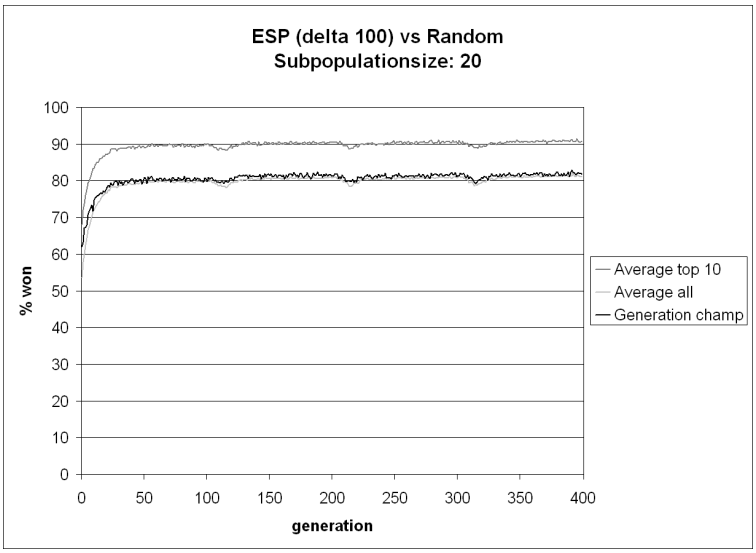

Figure 4.12: ESP, subpopulation size of 20, delta coding stagnation value 100

Comparing these results with the results without the delta coding show no real difference. In the end it does not seem to make a difference when delta coding is used or not for the smaller subpopulations. So ESP shows the best results when no delta coding is used and large subpopulation pools are used.

When comparing the *champion* percentage with the *average top 10* in the different graphs, the average top 10 score is higher than the score of the *generation champ*. This difference of 9 - 10% can be explained by the fact that the individuals are given fitness values based on 50 matches. After assigning the fitness there is a certain inaccuracy in the fitnesses assigned. After sorting the individuals, the ones with the highest (and also the most inaccurate) fitness make up the top 10. When having the best individual play 1000 matches as a more accurate fitness measure of that generation the *champion's* fitness is bound to be lower than the fitness of the *average top 10*.

The *champion* and *average all* show both graphs are closely together. The champion scoring better than the average of all evaluated individuals was something to be expected. The fact that both are close in win percentage shows that all evaluated individuals are not much spread out in win percentage. At the end of the ten experiments done with ESP without delta coding shows champions win percentage being close to one another. This, in contrary to SANE, shows ESP is much more reliable and stable in evolving.

ESP clearly performs better at Othello against a random opponent than SANE does. When using a subpopulatation size of 100 and no delta coding there is a difference between SANE and ESP of about 5%.

4.6 NEAT - Final experiments

4.6.1 Properties

As both other techniques, NEAT was also allowed to perform up to 2.000.000 evaluations per experiment. For NEAT the following parameters were used for the final experiments:

- Population size = 100 networks
- Add connection mutation rate = 0.2
- Add neuron mutation rate = 0.005
- Weight mutation rate = 0.01
- Speciation threshold = 0.6

4.6.2 Running Experiments

For NEAT ten experiments were done as well. *Figure 4.13* shows the results of these experiments.

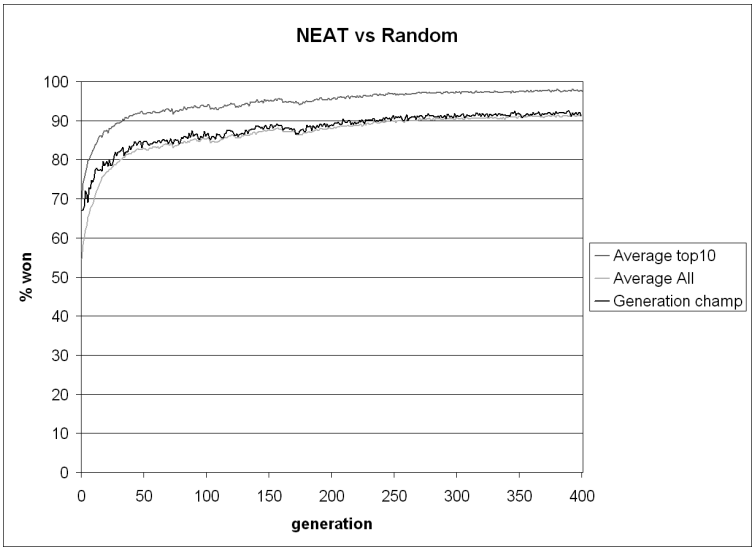

Figure 4.13: NEAT vs random, experimental results

NEAT reaches a win percentage of 90% at generation 200. NEAT does continue to improve to 92% in the end, and longer test runs might be needed to see how well NEAT can perform in the long run. Analysing the three graphs shows the *average top 10* to be at 98% wins, and the *champion's* win percentage to be slightly above the *average all* graph as is the case with ESP. Also at around generation 150 there is a dip in win percentage. This is the result of one of the ten experiments performing badly at that time.
Clearly NEAT performs better than both ESP and SANE.
Interesting to see were the amount of hidden units being evolved for the NEAT networks. The top 10 of individuals were started at 10 hidden units by default. The amount of hidden units was reduced to 2 and the amount of connections was reduced to 40-50 by the end of

the experiment while the win percentage kept rising. Apparently playing against a random playing opponent requires very few knowledge to be able to successfully beat it. This was confirmed when looking into the neural networks of the champions formed at the end of the experiments. Most champion networks had only two corner input nodes connected which suggests that a simple strategy that focuses on capturing some corners is enough to defeat a random opponent.

4.7 Deterministic opponents

Because of the small amount of hidden neurons and connections remaining in NEAT, some additional experiments were done to test if this was indeed the result of training against a random moving opponent. How will NEAT behave and learn when it plays against a player that uses a certain strategy?
Several experiments were performed to test this. The NEAT player plays against three different deterministic opponents who all play at a novice level. These players are explained in *chapter 2*.
Because NEAT is able to defeat the random player with a small neural network, it can probably also defeat the deterministic players, but it will probably take longer and will result in larger neural networks.

For these experiments the same parameters for NEAT were used as in the previous experiments against the random player.
Both the NEAT player and its opponent are deterministic players, so they will play the same games over and over. To avoid this, the first four moves of each player were made randomly. The number of different board states that may result from four random moves at the beginning of an Othello game is 244.

4.7.1 TD-Greedy opponent

The first opponent is the TD-Greedy player. This player uses a learned strategy which is not all clear. This strategy was learned by playing against random players and using off-line batch learning from worldclass tournament games.

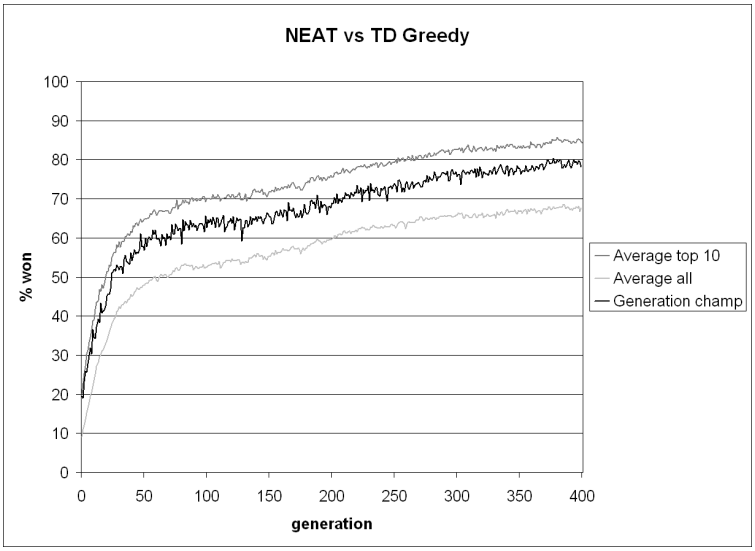

Figure 4.14: NEAT vs TD-Greedy

Figure 4.14 shows the results of this experiment. The graphs shows that NEAT is able to learn to defeat this opponent. Resulting in a 80% win and not even fully converged after

400 generations, NEAT is able to exploit the opponent's weakness.
The final neural networks all contained about 6 hidden hidden neurons which shows that a more sophisticated network is needed to defeat this player, than defeating a random opponent. Still only 6 hidden neurons can be seen as a small network.

4.7.2 Positional opponent

The second deterministic opponent is the Positional opponent. This player uses a simple but strong board evaluation to determine its moves. It can be beaten easily by using a mobility strategy to exploit the greediness of the player.
The results of this experiment are show in *Figure 4.15*.

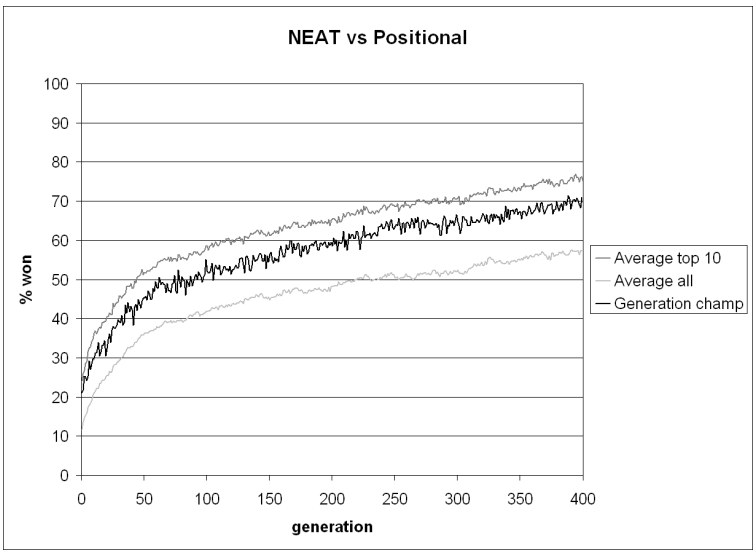

Figure 4.15: NEAT vs Positional

The graphs shows a slowly converging learning curve with a 70% win percentage after 400 generations. The curve has not converged yet, so it probably can learn to play even better over time.
It shows that NEAT is able to learn against the positional opponent, but it takes time. It learns slower against the positional opponent than against the TD-Greedy opponent. It is probably harder to find and exploit a weakness in the positional player.

The number of hidden neurons the neural networks have at the end of the last generation although is around 6, varying from 5 to 7.

4.7.3 Mobility opponent

The last deterministic opponent NEAT plays against is the Mobility player. Using a simple form of a mobility strategy this opponent is not hard to beat by an experienced player.
In *Figure 4.16* the results of this experiment are shown.

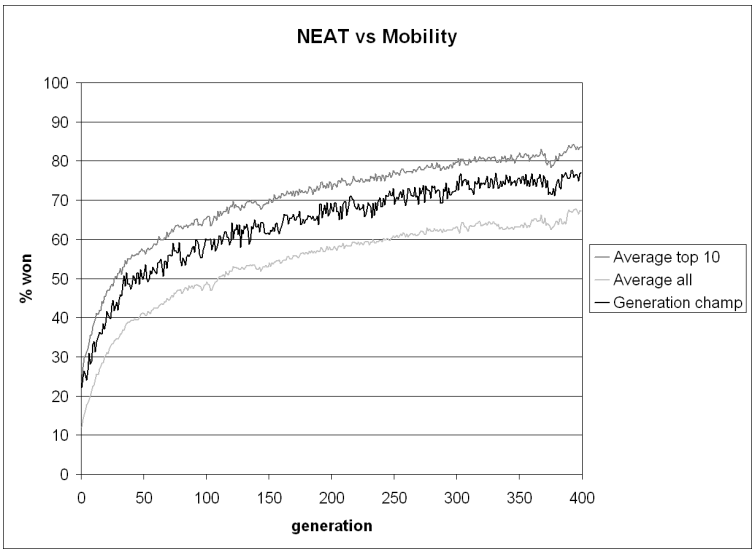

Figure 4.16: NEAT vs Mobility

The graphs is similar to the graph showing the positional opponent, although it seems to learn a little faster resulting in a 77% win percentage. NEAT is not converged yet at the end of the experiment, showing that it probably can learn to play even better.

At the end of the last generation the number of hidden neurons in the resulting neural networks is around 6, varying from 5 to 7.

The positional player seems to be the hardest player to learn against, although NEAT seems to be able to learn a very good game against each of the three deterministic players.
And because NEAT also is able to learn a good game against the random player, it is a potentially good neuroevolution technique to learn to play Othello.

Part II

Neuroevolution and Temporal Difference Learning

Chapter 5

TD Learning

In *chapter 3* and *chapter 4* genetic algorithms and Neuro-evolution techniques were used to learn to play Othello. In this chapter the *reinforcement learning* technique *temporal difference learning* will be explained and it will be used to learn Othello in the following chapters.
The temporal difference learning method has proven to be successful at learning to play Othello ([Jac05]). So this is an interesting technique to try and to combine with the successful neuroevolution technique NEAT. But because of a bug in the players used in [Jac05] (as explained in the next chapter) and for the sake of comparable results, several new experiments have been performed using reinforcement learning to learn Othello.

5.1 Reinforcement learning

Reinforcement learning ([Sut98]) is learning from interaction using a trial and error search, or more formal, mapping situations to actions to maximize the total reward when interacting in the environment.
The best way to understand reinforcement learning is to compare it with supervised learning. Where in supervised learning an agent is taught to respond to a given situation, in reinforcement learning the agent is not taught how to behave but it will find the best way to behave by using trial and error.
When interacting in an environment the reinforcement learning agent receives rewards for its behaviour. Normally a positive reward indicates good behaviour and a negative reward indicates bad behaviour. Using the rewards the agents can change its behaviour, its policy, to maximize its rewards in the long run.
One problem is that the agent rarely receives a reward after each action. It is more usual for the agent to take several actions before receiving rewards. Like in playing Othello, the player does not receive a reward after each move, because it will only know at the end of the game whether its sequence of actions was good. This is known as the *temporal credit assignment problem*: how can the actions the agent performed be rewarded if the reward only comes at the end of a sequence of actions?

Four main elements can be identified in a reinforcement learning system: a *policy*, a *reward function*, a *value function* and (optionally) a *model of the environment*.
The *policy* is the way the actor, or agent, behaves at a given time. It is a mapping from the states in the environment the agent perceives to the actions the agent can perform in the environment. In playing Othello the optimal policy is defined as selecting the right move in each state of the board.
The *reward function* defines the reward the agent gets in each state in the environment. It

is a mapping from a perceived state to a number, the reward. The objective of the agent is to maximize the total reward it receives in the long run. The reward is received by the agent from the environment. It is like experiencing pleasure and pain (positive and negative rewards) in a biological system.

The reward is used to alter the policy. When a low reward is received in a certain state after a certain action, then the policy might be altered so that the agent will avoid that action in the future.

The reward in playing Othello is win, draw or lose the game. So in Othello there is a delayed reward; the reward is given at the end of the game and has to be distributed over all moves the player did. How this is solved is explained in *paragraph 5.2*.

The *value function* defines what is good in the long run. The value function in a certain state is the total amount of reward an agent can expect to accumulate starting from that state. So it is a prediction of rewards and the purpose of estimating values is to achieve more reward.
In playing Othello the value function is used to determine which move to make given a certain state of the game. When several moves are possible, the value function estimates which move will have the highest total reward in the end of the game. The value function for each state is what the agent has to learn. But because Othello has a huge statespace, it is nearly impossible to visit all states and learn the value function for each state. So to generalize the state values a *function approximator* is used. In this case a Neural Network (see *paragraph 3.1*) is used as a function approximator. Using the current state of the board as input, it gives a value for that state. Similar boardstates will have similar values.
Another problem with learning the value function for all states is the trade-off between *exploration* and *exploitation*. When selecting a move, the player can always play greedily and select the move with the highest estimate (exploitation), but because the player may not be optimal, it could be not the best move after all. So once in a while the player has to select a different move to find out if that move is maybe better (exploration).
The solution used here is an ϵ-greedy method which means that with a probability 1 - ϵ a greedy move is performed and with probability ϵ a random move is performed. With a value of around 0.1 for ϵ and slowly decreasing this value to 0 over time, the player will explore new moves and in the end play a greedy strategy.

The fourth optional element of the reinforcement learning system is the *model*. The model mimics the behaviour of the environment so that a good prediction of the consequences of each action can be predicted. For playing Othello a model is not needed, because the game is fully observable and deterministic: it is exactly known what the next state is after each action. So a model is not used in this research.

5.2 TD(λ)

If one had to identify one idea as central and novel to reinforcement learning, it would be *temporal difference* (TD) learning ([Sut88]).

Temporal difference learning is a prediction method and as a prediction method, TD learning takes into account the fact that subsequent predictions are often correlated in some sense. In standard supervised predictive learning, an agent only learns from actually observed values: a prediction is made, and when the observation is available, the prediction is adjusted to better match the observation. The basic idea of TD learning is that it adjusts its predictions to match other, more accurate predictions. This procedure is a form of *bootstrapping*. So the goal is to make the current prediction for the current state more closely match the next prediction in the following time step.

When playing Othello, the player wants to predict the outcome of the game at every game state. Of course this is nearly impossible at the beginning of the games, but can be done pretty accurate towards the end. Using TD learning, the player has to learn to predict the outcome by matching the prediction more closely to the next prediction in the next state. The last gamestate contains the final reward when the game ends.
So how can the predictions for all gamestates (up to about 30) be adjusted when there is only 1 final reward in the last gamestate?
This is accomplished by using *eligibility traces*. When using a simple TD method, only the immediately preceding gamestate is updated and not all earlier predictions. But for playing Othello it is necessary that all predictions are updated. Eligibility traces do this by providing a short-term memory of the previous gamestates. They are usually implemented by an exponentially-decaying memory trace, with decay parameter λ. The TD methods that uses these eligibility traces with the decay parameter are called TD(λ) methods.
TD(λ) and $\lambda = 0$ corresponds to updating only the immediately preceding prediction, and $\lambda = 1$ corresponds to equally updating all the preceding predictions.
But for Othello, the early predictions in the game are probably not that accurate while the predictions towards the end can be very accurate. That is why a value of λ between 0 and 1 is probably the best value. This way the predictions in the end of the game are updated stronger than the predictions in the beginning of the game.

The formulae in *figure 5.1* are used in the Othello implementation.

$$\boxed{\begin{array}{l} V'(s_\top) = R_\top \\ V'(S_t) = \gamma V(s_{t+1}) + r_t + \gamma\lambda(V'(s_{t+1}) - V(s_{t+1})) \end{array}}$$

Figure 5.1: Calculation of TD(λ) statevalues

Definitions used in these formulae:

$\top$	: Terminal state, i.e. last state of the episode
$R_\top$	: Final reward at the end of the episode
s_t	: state s at timestep t in episode
$V'(s_t)$	: The new calculated statevalue of state s at timestep t
r_t	: the reward at timestep t (always zero in this research)
γ	: discount factor which in this research is 1
λ	: decay factor of weight distribution of the final reward

In this definition, based on definitions in [Sut98], the value for $r_t = 0$ and $\gamma = 1$. The value 0 for r_t is used because there is no reward during the game. Only at the end of the game, the player receives a reward: 1 for a win, 0.5 for a draw and 0 for a loss. The value for $\gamma = 1$ because no discounting is necessary for Othello

5.3 TD(λ) and Othello

To learn to play Othello using TD(λ) it is necessary to define episodes and states in the gameplay and how rewards are given.
To compare TD(λ) with the other learning techniques also 2.000.000 evaluations are allowed, which means that a sequence of 2.000.000 games of Othello can be used to learn a good strategy. For this research each game of Othello is one episode and each game, or episode, consists of several states. A reward is given at the end of each game, which can be 1 for winning, 0 for losing and 0.5 for a draw.
As a function approximator for the value function a neural network is used. To learn the value function the formulae in *figure 5.1* are used.

Because the game is fully observable it is possible to learn from the opponent's moves as well as from the player's own moves.
An off-line learning method is used, which means that during one episode all states plus their statevalues are stored and after the episode the learning process starts.
Figure 5.2 shows the pseudocode for the algorithm used to play a game and store all statevalues.

```
Initialize s ← initial boardstate
do:
    if player's turn:
        with probability 1 − ε
            a ← action given by NN for s
        else:
            a ← random legal action
        s' ← state after performing action a
        V_p(s') ← statevalue of s' given by NN
        PerformAction(a)
    if opponent's turn:
        a ← action opponent performed
        s' ← state after performing action a
        V_o(s') ← statevalue of s' given by NN
    s ← s'
until end of game is reached
V'_p(s) ← final statevalue is the final reward for player
V'_o(s) ← final statevalue is the final reward for opponent
```

Figure 5.2: TD(λ) Othello episode pseudocode

The algorithm starts with the initial board. Then for each turn the resulting state of the

chosen move is stored with its state-value. So when it is the player's turn, the player selects an action a based on the ϵ-greedy method, so it can be a greedy move or an exploration move, and it determines the state s' the board will be in after performing the action a. Then using the neural network, the state-value of that state s' is calculated and stored in $V_p(s')$.
For the opponent's move the resulting state s' is evaluated using the neural network to determine the state-value of that state and stored in $V_o(s')$.
After the episode ends the rewards are set to the final state-values of both the player and the opponent. If the player wins, it receives 1 and the opponent receives 0. If the opponent wins the player receives 0 and the opponent receives 1. In case of a draw, both players receive 0.5 as a final reward.

To really learn from the stored episode the algorithm as shown in *figure 5.3* is used. This algorithm uses a sequence of 2.000.000 episodes to learn to play Othello learning from both the player and the opponent.

$\epsilon \leftarrow 0.1$
ϵ-decrease $\leftarrow \epsilon / 2.000.000$
for $i \leftarrow 1$ to 2.000.000:
    $PlayEpisode()$
    $s_p \leftarrow s_{\top p}$
    $s_o \leftarrow s_{\top o}$
    $BackpropNN(s_p, V'_p(s_p))$
    $BackpropNN(s_o, V'_o(s_o)$
    do:
        $s_p \leftarrow$ previous state of player
        $s_o \leftarrow$ previous state of opponent
        $V'_p(s_p) \leftarrow \gamma V_p(s_p) + \gamma\lambda(V'_p(s_p) - V_p(s_p))$
        $V'_o(s_o) \leftarrow \gamma V_o(s_o) + \gamma\lambda(V'_o(s_o) - V_o(s_o))$
        $BackpropNN(s_p, V'_p(s_p))$
        $BackpropNN(s_o, V'_o(s_o)$
    until reached first state of episode
    $\epsilon \leftarrow \epsilon - \epsilon$-decrease
    if i modulo 2000 = 0:
        $Play1000Games()$
        $playerstrength \leftarrow$ percentage of games won

Figure 5.3: TD(λ) from episodes pseudocode

The algorithm starts with initializing the ϵ value to 0.1 and the ϵ-decrease to ϵ / 2.000.000. So the value of ϵ decreases from 0.1 to 0 over 2.000.000 games. This means every new episode the chance of exploration decreases and the player will get more greedy towards the end of the 2.000.000 games.
After initialisation the sequence of games starts with first playing the game and storing state-values of each state for both player and opponent and determining the final reward using the algorithm in *figure 5.2*.

Then after the end of the game the last states of both players, the terminal states $s_{\top p}$ with their state-value (which is equal to the final reward of the game) are learned using the backpropagation function in the neural network.
After that, all previous states are learned using the same backpropagation method and using the formulae showed in *figure 5.1*.
This will go on for 2000 games and then the player will use its current neural network to play 1000 games against its opponent without learning and with a greedy action selection to determine the strength of the player at that moment. The percentage of games won determines the strength of the player and can be compared with other learning techniques. Because every 2000 games the strength of the player is determined, a graph can be produced with 1000 points of the strength of the player over a sequence of 2.000.000 games.

The next chapter describes the implementation of the experiments performed using these algorithms.

Chapter 6

TD(λ) Experiments

6.1 Implementation

In [Jac05] several experiments using TD(λ) have been performed. One big problem with the results in that research is that the random player that was used, contained a bug. This bug resulted in random selection from all possible moves minus 1. So if in a state there were 5 possible moves, it picked a move from the first 4 possible moves. This resulted in a random player that was not capable of using all possible moves, and thus covered less possible gamestates and in the end this player was easier to defeat.
Therefore new experiments against both random and deterministic players were performed in this research using correct working randomness where needed.
The basic implementation from [Jac05] was used for the TD(λ) player with minor changes and fixes.
The TD(λ) player uses a fully connected feedforward neural network with a sigmoid activation function to return a value between 0 and 1. It keeps track of both own moves and the opponent moves. At the end of each game, the neural network is updated for both own gamestates and opponent's gamestates using the correct rewards and backpropagation. This way the player can learn from both its own gameplay and the opponent's gameplay.

For all experiments in this chapter 2.000.000 evaluations are allowed, which means that the TD(λ) player can play 2.000.000 games against its opponent to learn a good strategy. To keep track of the learning progression after each 2000 games the TD(λ) player plays 1000 games against the opponent using a greedy policy without learning. The win percentage of these 1000 games is used as the strength, or fitness of the player at that moment. Using the win percentage of 1000 games the score can be compared with the neuroevolution experiments and the experiments in the next chapters. An ϵ-greedy method is used for the trade-off between exploration and exploitation as explained in the previous chapter.
Each experiment was repeated 10 times to make a good average.

6.2 Random opponents

To find out how well a TD(λ) can learn against a random player, several experiments were performed with varying parameters. First several values for λ are tested against the random player. Then the neural network learning rate, α, is tested and in the end different sizes of the neural network are tested. Because NEAT showed that it can learn a good strategy with a very small neural network, it is interesting to find out whether TD(λ) is also able to learn a good strategy with a minimal size network.

6.2.1 λ experiments

The results in [Jac05] showed a good value of λ is 0.9. But because of the bug in this research, as explained in the first paragraph, new experiments are necessary.
This time the values for λ 0.6, 0.8, 0.9 and 1.0 are tested against the random player. The results are shown in *Figure 6.1.*
The value for λ, the learning rate of the neural network, is set ot 0.01 for these experiments. The neural network contains 20 hidden units.

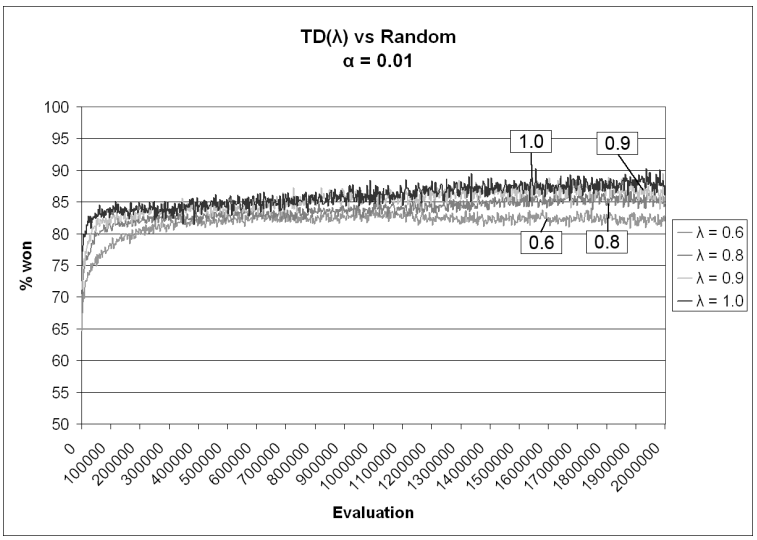

Figure 6.1: TD(λ) vs Random with different values for λ

As expected the lower values for λ do not produce the best results. The λ of 0.9 and 1.0 produce the best results, where 1.0 seems to perform even a little bit better than 0.9, although the difference is minimal. So the value of 0.9 for λ seems a good value. The value 1.0 is not chosen, because 1.0 and 0.9 show minimal difference and a value of 0.9 is a good example of TD(λ) where a value of 1.0 is in fact an (improved) Monte Carlo method, a method not covered in this research ([Sut98]).
So a λ value of 0.9 will be used in the following experiments.

The best values for λ have a final score around 87%, which is a good score. All values for λ seem to converge after around 1.500.000 evaluations.
Compared to NEAT, the TD(λ) scores a little less in the end. NEAT reaches a score of about 92% after 2.000.000 evaluations. Both are able to learn pretty well against a random opponent.

6.2.2 α experiments

The α is the learning rate of the neural network. A small learning rate needs more learning for a good solution, but a high learning rate may result in sub-optimal results. In this experiment α values of 0.1, 0.01 and 0.001 are tested. The λ used is 0.9.

In [Jac05] a value of 0.35 for α was used which seems very high. The good results in that research may be caused by the fact that the random player did not cover all gamestates, so the neural network had to learn from less different gamestates.
The results are shown in *Figure 6.2*.

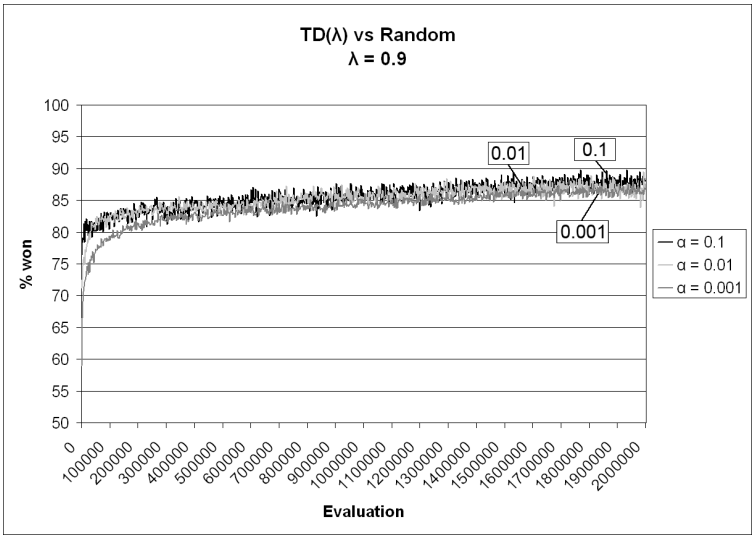

Figure 6.2: TD(λ) vs Random with different values for α

The graph shows that a value 0.1 or 0.01 for α works best. Both showing the same learning speed and the same results in the end: a win percentage around 87%.
As expected the value of 0.001 for α learns slower, but after the 2.000.000 games it looks like it has not even converged all yet, where the 0.1 and 0.01 converged after around 1.500.000 games. The final results of 0.001 is a little lower, but all seem to converge to the same value in the long run. The big difference is the speed of learning. An α with value 0.001 learns slower than 0.1 and 0.01.
The fact that all values for α result in almost the same outcome is probably because of the many different states the players evaluates. Because a random opponent does not use a strategy but random moves, there are no states that are visited that often. So no overfitting of the neural network takes place.
The results of different values for α when playing against a deterministic player will probably differ more, because a lot of gamestates are visited more than once so a bigger α will probably result in a sub-optimal result, as where low values will learn slower, but better in the end.
Experiments with deterministic opponents are performed in *paragraphs 6.3.2* and *6.3.2*.

6.2.3 Network size experiments

In *chapter 4* it was shown that a very small neural network was able to play a good strategy against a random player. This was shown using NEAT to evolve the neural networks. To find out whether TD(λ) is able to learn using a small neural network, different network sizes are tested against the random player using the best α and λ from the previous experiment, 0.01 for α and 0.9 for λ.

The number of hidden units tested in this experiments are 2, 5, 10 and 20, where the results for 20 are taken from previous experiments.
The results are shown in *Figure 6.3*.

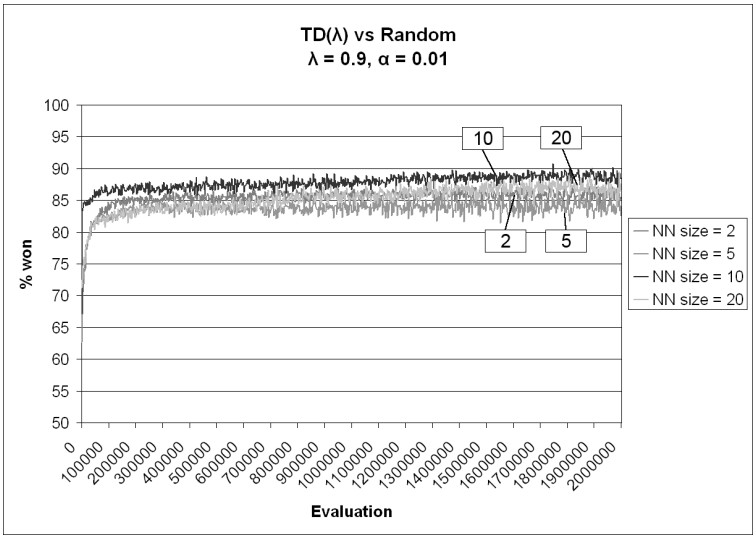

Figure 6.3: TD(λ) vs Random with different NN sizes

All networks are able to learn pretty well. All end up with a final score from 84% to about 88%. The network with 10 hidden units seems to be the best. It has the highest learning speed and ends up with the highest score.
The very small networks perform worse. Both a size of 2 and 4 end up around 85%.

Although a bigger neural network can do a better approximation, against a random player a small network is enough to win. This means that combining TD(λ) with NEAT has potential, because NEAT also ended up with small networks in the end.

6.3 Deterministic opponents

The TD(λ) method was able to learn to play Othello very well against a random opponent. This was to be expected because it was proven before in [Jac05].
But how well will the TD(λ) method perform when learning against a deterministic player? Will it learn to exploit the weakness of its opponent?
In this section the TD(λ) is tested against the TD-Greedy opponent, Positional opponent and the Mobility opponent. All three opponents are explained in *chapter 2*.

6.3.1 TD-Greedy opponent

The TD-Greedy player uses a learned strategy which is not all clear. This strategy was learned by playing against random players and using off-line batch learning from worldclass tournament games. It plays a good game against novice players.

The TD(λ) is tested using $\lambda = 0.9$ and a neural network size of 20 hidden neurons. This large size of the neural network is chosen because a more sophisticated function approximation may be needed to learn a good strategy.
An ϵ-greedy method is used for the trade-off between exploration and exploitation.
Both a value 0.01 and 0.001 for α are tested.
The results are shown in *Figure 6.4*.

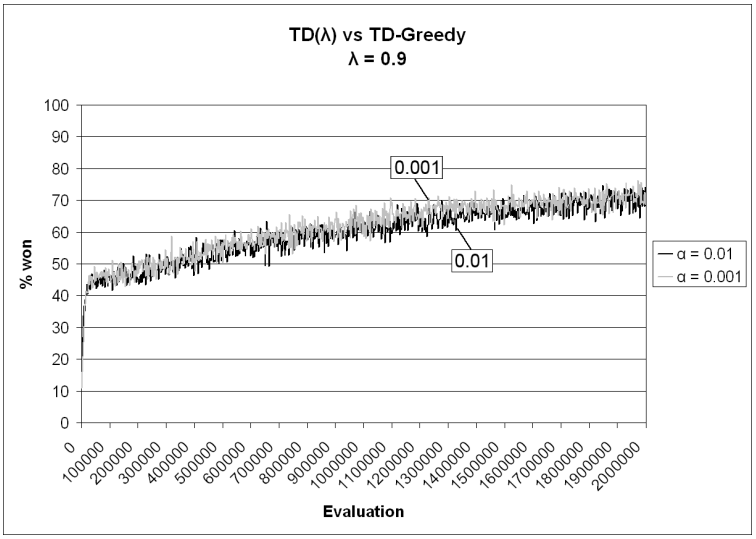

Figure 6.4: TD(λ) vs TD-Greedy with different values for α

First thing to notice is that there is not really a difference between the two values of α. Both 0.01 and 0.001 show the same learning curve.
The next thing to notice is that the player is able to learn to defeat the TD-Greedy opponent, resulting in a 70% win percentage after 2.000.000 games. And because the player has not converged yet, it could be even higher in the long run.
In the very beginning of the sequence of games, the TD-Greedy opponent still is strong, beating the TD(λ) player most of the time. But after 250.000 games the TD(λ) player has learned so much, it can defeat the TD-Greedy opponent half of the time. From then the win percentages keeps increasing.
So like NEAT, TD(λ) is able to learn to defeat the TD-Greedy player, although NEAT seems to learn faster.
Combining these two techniques might result in even faster learning to defeat this opponent. In the next chapters this will be tested.

6.3.2 Positional player

The positional opponent uses a simple board evaluation and will focus on short time reward by trying to capture the corner positions and the edge positions as soon as possible. A positional player can easily be defeated by a good mobility player, but can a TD(λ) player learn from the weakness of a positional player and exploit that knowledge?

The TD(λ) is tested against the positional player using $\lambda = 0.9$ and a neural network size of 20 hidden neurons.
Here also an ϵ-greedy method is used for the trade-off between exploration and exploitation.
Both a value 0.01 and 0.001 for α are tested.
The results are shown in *Figure 6.5*.

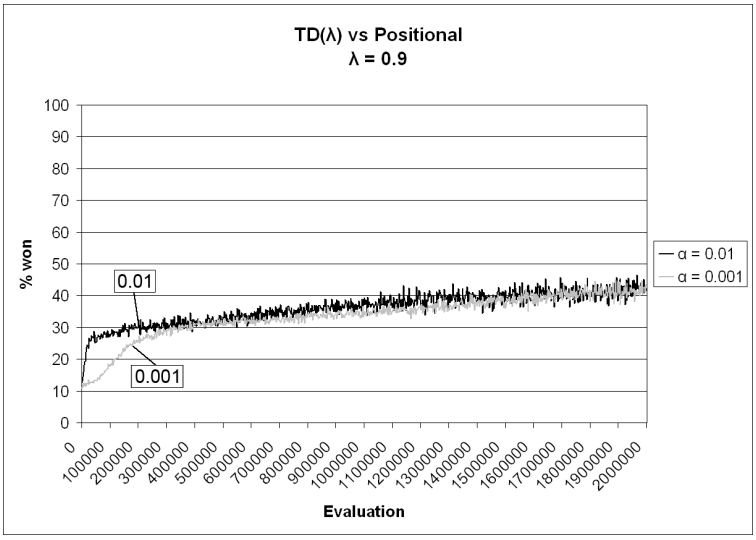

Figure 6.5: TD(λ) vs Positional with different values for α

The first thing to notice is that TD(λ) is not even able to win more than half of the games after learning for 2.000.000 episodes.
As expected the α of 0.001 learns slower than 0.01, but after 500.000 games the learning curves are the same. Both are not able to learn to defeat the positional opponent sufficiently after 2.000.000 games.
However, both 0.01 and 0.001 are not yet converged so in the long run they both might win more than half of the games. But apparently, 2.000.000 games is not enough to learn against a positional opponent.
Maybe it is because of too few exploration moves. There is too little time to do more research on this opponent.

As seen in *chapter 4* NEAT was able to learn to defeat the positional opponent reaching a score of 70%. And after playing 2.000.000 games NEAT still has not converged.
So what will happen when NEAT and TD(λ) are combined and learning against the positional opponent?
Probably the results will not be better than NEAT alone and maybe the TD(λ) will even have a negative influence on the final results. In the next two chapters this will be tested.

6.3.3 Mobility player

The mobility opponent uses a simple form of the mobility strategy. The way this player works is explained in *chapter 2*.

How well will the TD(λ) perform against this opponent?
The results against the positional opponent were a little disappointing, but maybe the TD(λ) can exploit the weakness of this opponent better. The weakness is that this opponent does not really use a good mobility strategy. It does not look ahead more than one move so it is hard to really use a mobility strategy. The TD(λ) player might be able to use a relatively simple strategy to exploit that weakness.

Here the same value of λ, 0.9 and the same neural network size of 20 hidden units are used as with the experiment against the positional opponent.
Also both 0.01 and 0.001 are used as values for α.
An ϵ-greedy method is used for the trade-off between exploration and exploitation.
The results of the experiments are shown in *Figure 6.6*.

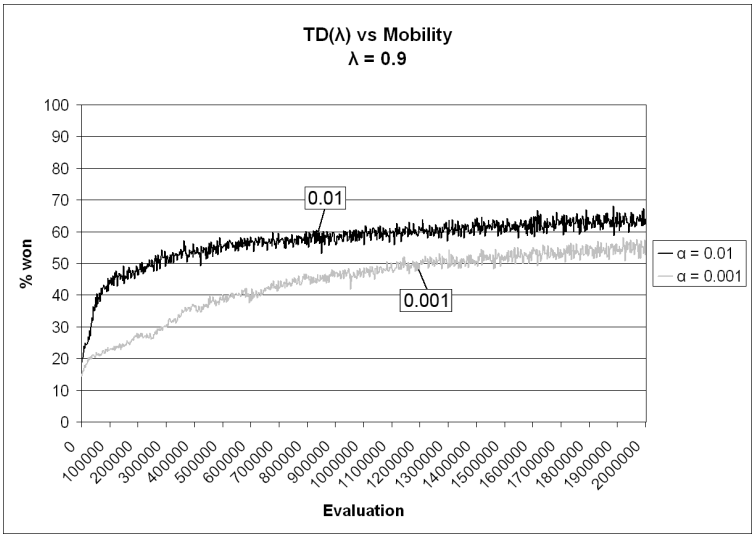

Figure 6.6: TD(λ) vs Mobility with different values for α

The first thing to notice is that both α 0.01 and 0.001 are able to defeat the mobility player more than half of the time in the end.
The α of 0.01 seems to learn better, but both are still not converged after 2.000.000 games, so they might end up with the same results in the end.
However, for these 2.000.000 games, the value of 0.01 seems to outperform the other. After 250.000 games it defeats the mobility opponent more than half of the time, and in the end it is about 65%.

When compared to NEAT in *chapter 4* it does not perform that good. NEAT has a score of 76%, which is significantly better than TD(λ).
So also for this mobility opponent the question rises: how will NEAT combined with TD(λ) perform against this mobility player?
Probably the same as against the positional opponent: no positive influence, and maybe even a negative influence on the end results.
In the next two chapters this will be tested.

Chapter 7

NEAT-TD(λ)

7.1 Combining NEAT and TD(λ)

As can been seen in the previous chapters, both NEAT and TD(λ) are able to learn to play Othello. NEAT seems to learn faster and ends up with better results after 2.000.000 evaluations, but both techniques are able to learn.
The way NEAT and TD(λ) learn are completely different. TD(λ) starts with a neural network and from there changes it slightly after every new game, thus taking small steps in the search space in the search for an optimum. NEAT starts a generation with several (100 in this research) individual neural networks all spread out over the search space and uses genetic algorithms to create new networks which are not necessary close to earlier networks in the search space. So the NEAT method uses a more or less global search, where TD(λ) uses a kind of local search in the search space.
As NEAT uses an intelligent random global search in the search space, thus taking large steps in the search space, this search method may very well be improved by using a local search after every global move in the search space. A big step in the search space can end up close to an optimum, so with a local search starting from there, it can be taken closer to that optimum.

So combining a global search, NEAT, with a local search, TD(λ), seems to have potential theoretically. The question is: how can it be achieved?
In this chapter a method to combine NEAT and TD(λ) to a method called NEAT-TD(λ) is described and in the next chapter several experiments are performed to test this combination.
What NEAT-TD(λ) basically does is using NEAT to evolve the neural networks by evolving the topology and weights as described in earlier chapters and then extend this method by also updating the weights of the connections in the neural network using TD(λ). This is described in detail in *paragraph 7.3*.
To achieve this the used implementation of NEAT, Anji, had to be improved and extended, because it was not capable of including TD(λ). This is described in detail in *paragraph 7.2*.

It is obvious that it is not desirable to use TD(λ) to learn from 2.000.000 evaluations for every individual in every generation. Because of time constraints, TD(λ) can only learn from a limited amount of games. To create comparable results, it is necessary to use a total of 2.000.000 evaluations for the whole experiment. This means that TD(λ) will have to use the 50 games that NEAT uses for each individual to learn. This can be extended by using a form of batch learning using games from other individuals as well. How this is done is explained in *paragraph 7.3*. The big question is: is this limited amount of games for TD(λ)

enough to improve the results of NEAT?

There is not that much research done in combining NEAT with reinforcement learning. In [Whi06] a research combining NEAT and Q-learning, a TD-method, is described. This new technique, which they call NEAT-Q, *evolves* individuals that are better able to *learn*. This method is tested in two domains: a mountain car task and a server job scheduling. The authors conclude that their method is better than the two individual techniques.

Q-Learning is not used here to play Othello, so their claim cannot easily be extended to this research. Therefore experiments using Othello have to be performed. Looking at the results of earlier experiments, a small improvement can be expected. Maybe not all experiments will show better results. For example, the results of TD(λ) against the Positional opponent were not good, so combining NEAT and TD(λ) may result in a lower win percentage than NEAT alone. Individual expectations for each experiment are described in the next chapter.

7.2 Backpropagation

For this research the Anji implementation of NEAT is used as described in *chapter 4*. It uses a neural network for the function approximation, but the problem is that this implementation of the neural network does not support backpropagation. To use TD(λ), backpropagation is necessary.
Because of the complexity of the implementation of Anji, it is not possible to add backpropagation to this implementation. So a whole new implementation of the neural network had to be created that is able to handle all the unusual topologies that NEAT can produce, and is able to perform backpropagation to update the weights of the connections.
One of the problems with the Anji implementation was the way the feedforward activation works. The activation uses several *activation sweeps* through the network until all neurons have been activated. With each sweep, all neurons, starting with the input neurons, are activated, and with every sweep more neurons are activated. This is done because of the unusual topologies that allow neural networks to have connections skipping layers. So a neuron in layer 1 can connect to a neuron in layer 5. This makes a feedforward activation difficult, because there are no real layers, so it cannot activate layer by layer. And using several activation sweeps made the Anji implementation relatively slow.

When designing a new implementation for the neural network, this problem was solved by creating an order of neurons in which the neurons have to be activated. When using a correct order, all neurons can be activated in one sweep.
In *figure 7.1* a small neural network is shown with 2 input neurons and 1 output neuron and several hidden neurons. This is an unusual topology which can be created by NEAT.
The neurons 1 and 2 are the input neurons and neuron 6 is the output neuron. The feedforward activation of Anji will in its first sweep activate neuron 1 and 2, because these are the only neurons with a value. This results in an activation value for neuron 3 and a part of the activation for neuron 4 and 6. So in the next sweep neuron 1, 2 and 3 are activated, resulting in an activation value for neuron 4. In the following sweep neuron 5 will have a value and in the last sweep the output neuron will have a value. So it takes 4 sweeps

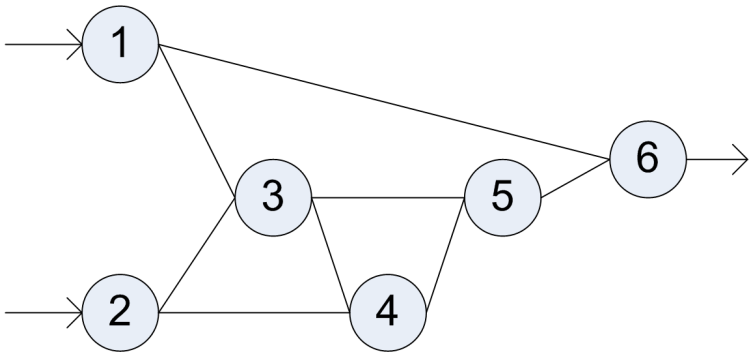

Figure 7.1: A simple neural network with an unusual topology

to produce an output value for this simple neural network. This is very time consuming, because in Othello for every board evaluation a feedforward of the network is needed.

So the new neural network creates an order of neurons by looking how the neurons are connected. Assuming no cycles are possible and a neuron never connects to a neuron in a lower layer, an order can be created. These assumptions are justified by the Anji implementation.

The new neural network creates a list which starts with all the input neurons. Then for each neuron it will look at its incoming connections and then place the neuron in the correct position in the list, resulting in an ordered list. This order can be used for a one-sweep activation of the network.

The order of the given network will be 1 2 3 4 5 6 or 2 1 3 4 5 6.

With the use of this ordered list, the problem of backpropagation for these kind of topologies can also be solved. The hard part of backpropagation with these types of topologies is the order in which the neurons should be updated. This can be done with several sweeps as Anji does with the feedforward, but by using the inverse order of the feedforward list, it can be done in one sweep.

The feedforward of this new neural network has been tested against the Anji neural network with playing Othello to ensure it behaves exactly the same as the Anji implementation of the neural network. After playing millions of games there were no differences in the board evaluations.

The backpropagation was tested by creating several different topologies and then learning the XOR function which worked correctly.

Because this neural network used only one sweep for the feedforward, playing Othello is now significantly faster. With the new neural network, learning NEAT is about 400% faster which is a huge improvement.

7.3 Evaluation function

To combine the two techniques the implementation of NEAT as described in *chapter 4* is used and is extended with the TD(λ) implementation as described in *chapter 5*. So there still are 100 individual neural networks that are evolved, but before evolving to the next generation, all individuals use TD(λ) to improve themselves.
The global algorithm for NEAT-TD(λ) is shown in *figure 7.2*.

```
Initialize generation
for i ← 1 to 400:
    for each individual:
        Store50Games()
    for each individual:
        LearnFromGamedatabase()
    for each individual:
        Play50Games()
        p ← percentage games won
        fitness ← (p+δ_rms*100)/2
    Play1000GamesWithChampion()
    playerstrength ← percentage of games won
    EvolveGeneration()
```

Figure 7.2: NEAT-TD(λ) algorithm pseudocode

The algorithm starts off with initializing a random population of networks. The same parameters of NEAT are used as in earlier experiments.
This algorithm is explained in detail in the next paragraphs.
In short this algorithm works as follows: first 50 games are played by each individual and all gamestates of these games are stored in a gamedatabase.
Then each individual can learn a certain number of games from this gamedatabase using the TD(λ) method.
After that, the improved neural networks are used to play another 50 games for each individual to determine its fitness. The fitness is now a combination of the win percentage and the TD-error which will be explained in the next paragraphs.
Then the champion of the generation, ie the neural network with the highest fitness, can play a 1000 games to determine its strength.
Then at the end of the generation, the genetic operations are used to generate new offspring, and then the whole algorithm starts from the beginning.
This will go on for 400 generations, the same as in earlier NEAT experiments, giving NEAT-TD(λ) 2.000.000 games to play and learn from.

7.3.1 Storing gamestates

In the earlier used implementation 50 games were played by each individual to determine the fitness. For NEAT-TD(λ) also 50 games are allowed for each individual to determine the fitness. But 50 games might not be enough for the TD(λ) method to learn. That is why

all individuals play 50 games, and all those games and their gamestates are stored creating a database of gamestates to learn from.

Figure 7.3 shows the pseudocode for the algorithm that stores the 50 games of each individual.

```
for i ← 1 to 50:
    g_p ← empty set of player gamestates
    g_o ← empty set of opponent gamestates
    do:
      if player's turn:
        a ← action given by NN for s
        s_p ← state after performing action a
        StorePlayerGamestate(s_p, g_p)
        PerformAction(a)
      if opponent's turn:
        a ← action opponent performed
        s_o ← state after performing action a
        StoreOpponentGamestate(s_o)
    until end of game is reached
    r ← final reward of the game for the player
    SaveAllgamestatesToFile(r, g_p, g_o)
```

Figure 7.3: NEAT-TD(λ) Store50Games function pseudocode

The algorithm shows that for every of the 50 games an empty set of gamestates for the player and for the gamestates of the opponent are created. Because the player must be able to learn from both itself and from its opponent, both the moves of the player and the opponent are stored by storing all the gamestates they both visit.

During the game the player will use a greedy policy and select the move by using the neural network for the board evaluation. It will then store the resulting gamestate in the set for the player.

Whenever the opponent makes a move, the resulting gamestate of that move is stored in the set of gamestates for the opponent.

After the game ends, the reward is calculated, using 1 for a win for the player, 0 for a loss and 0.5 for a draw. All gamestates are then stored in a file. All player gamestates are stored with the reward the player received. Alls opponent gamestates are inverted, by flipping all black and white pieces, so it is as if the player had performed all the moves instead of the opponent. The reward is also inverted, so if the player had lost the game, the opponent won resulting in a reward of 1 for the opponent gamestates.

The result for each game is a set of gamestates with the player's moves and its final reward and a set of inverted gamestates of the opponent with the inverted reward. This way the player can later select any of those sets to learn from as if it were its own gamestates and reward.

7.3.2 Learning from gamestates

When all 50 games of each individual are stored, the player can start learning from this database of games. There are 100 individuals playing 50 games and from each game 2 sets of gamestates are stored, resulting in 10.000 sets of gamestates to learn from.
As shown in the algorithm in *figure 7.2* the fitness is not only determined by the win percentage of 50 games, but also uses the TD-error, δ_{rms}. This is the Root Mean Square Error and it is calculated during the learning process.
The definition of the δ_{rms} is given in *figure 7.4*.

$$\delta_{rms} \leftarrow \frac{1}{g} \sum_{i=0}^{g} \sqrt{\frac{1}{s_i} \sum_{j=1}^{s_i} \delta_{ij}^2}$$

Figure 7.4: Calculation of the Root Mean Square Error

Where:

g : number of games
s_i : number of gamestates in game i
δ_{ij} : error of learning gamestate j of game i ($\delta \leftarrow (V' - V)$)

It uses the difference between the estimated value of the value function and the real value based on the reward. For each gamestate in the game this difference is calculated and squared. For the whole game the error is the square root of the sum of these squares divided by the number of gamestates.
The total error δ_{rms} of the player is the average error of all games.

This error calculated during the learning process as can be seen in *figure 7.5* which shows the algorithm of the learning process.
The algorithm starts with retrieving two sets of gamestates from the stored games of the current generation. A set with the 5000 player games and a set with the 5000 opponent games. The TD-error of the player is initialized as 0.
Then a sequence of 500 learning cycles is started, switching between learning from the player's gamestates and the opponent's gamestates.
So every cycle a game is randomly selected from the player games or the opponent games. The final reward r and the terminal state s are retrieved. Then the $V'(s)$ is set to r and the neural network is updated with that state-value.
Then for each previous state s $V'(s)$ is calculated using the formula in *figure 5.1* explained in *chapter 5* until the first state of the game is reached.
During this learning process the TD-error is calculated as well. At the end of the cycle of 500 games, the player has learned 250 own games and 250 opponent games and it has calculated its TD-error.

$g_p \leftarrow$ set of games with gamestates of player
$g_o \leftarrow$ set of games with gamestates of opponent
$\delta_{total} \leftarrow 0$
for i $\leftarrow$ 1 to 500:
    $\delta_{game} \leftarrow 0$
    if i modulo 2 = 0:
        $g \leftarrow$ random game from g_p
    else:
        $g \leftarrow$ random game from g_o
    $r \leftarrow$ final reward of the game g for the player
    $n \leftarrow$ number of gamestates in game g
    $s \leftarrow$ terminal state of game g
    $V'(s) \leftarrow r$
    $BackpropNN(s, V'(s))$
    for j $\leftarrow n$ to 0:
        $s \leftarrow$ previous gamestate
        $V(s) \leftarrow FeedForward(s)$
        $V'(s) \leftarrow \gamma V(s) + \gamma\lambda(V'(s) - V(s))$
        $\delta \leftarrow V'(s) - V(s)$
        $\delta_{game} \leftarrow \delta_{game} + \delta^2$
        $BackpropNN(s, V'(s))$
    $\delta_{total} \leftarrow \delta_{total} + \sqrt{\frac{\delta_{game}}{n}}$
$\delta_{rms} \leftarrow \frac{\delta_{total}}{500}$

Figure 7.5: NEAT-TD(λ) learning from gamestates database pseudocode

7.3.3 Determine fitness

After the TD(λ) learning process the neural networks are ready to play another 50 games to determine the final fitness. During the learning process the TD-error is calculated for the error, and this will be half of the final fitness. The other half is determined by the win percentage of the 50 games.
So the player plays 50 games against the opponent using a greedy policy with the newly learned neural networks. Then the final fitness is calculated using the formula in *figure 7.6*

$$fitness \leftarrow \frac{p + \delta_{rms} * 100}{2}$$

Figure 7.6: Calculation final fitness

Where:

$fitness$	: the final fitness of the player. A value between 0 and 100
p	: win percentage after playing 50 games
δ_{rms}	: The root mean square TD-error of the player

The δ_{rms} is a value between 0 and 1 so the final fitness is a value between 0 and 100.
This fitness value is used to sort the individuals for the genetic reproduction in the NEAT algorithm. The player with the highest fitness also plays 1000 games against the opponent to determine the strength of the generation champion. The win percentage of the player after these games is the strength of the player and is shown in the graphs in the next chapter.

The next chapter covers the experiments performed using the NEAT-TD(λ) combination.

7.4 Number of evaluations

When looking at the used algorithm here for NEAT-TD(λ) one can say that there are 100 games played for each individual, resulting in 4.000.000 evaluations after 400 generations, instead of 2.000.000 in previous chapters.
It is true that each individual plays 100 games, but only 50 games are used to determine the fitness. The first 50 games are only used to create a gamedatabase. A solution to really use 50 games for each individual could be by using the games from the previous generation to learn from with the TD(λ) method, instead of playing them each generation. This will result in really 50 games in each generation, and no TD-learning in generation 1, because of the lack of a gamedatabase at the start.
The results of learning from games of the previous generation would have the same results, because the games are only used as training data, and training data from 1 generation back will not differ that much from the data in the current generation.

This solution will end up with the same results and will probably be quicker because less games are played. The problem is that this solution was found after performing the experiments and there was no time left for new experiments. So the experiments in the next chapter use 4.000.000 evaluations, but have the same results as when 2.000.000 evaluations had been used.

Chapter 8

NEAT-TD(λ) Experiments

8.1 Experiment parameters

To compare the experiments of NEAT-TD(λ) with previous results, the player is also allowed 2.000.000 evaluations. Using 100 individuals each generation and 50 games for each individual, the player can play 400 generations, the same amount as in earlier neuroevolution experiments.
For these experiments the Anji implementation of NEAT, as used in chapter *chapter 4*, is combined with the TD(λ) implementation of *chapter 6*.
The implementation of Anji was extended so it had support for backpropagation and was able to include the TD(λ) method. This is described in the previous chapter.
As stated in the previous chapter, a form of batch learning is used. Each generation all gamestates of all games played are stored in a file so the TD(λ) algorithm can randomly choose games to learn.
All games are played with a greedy policy using the neural network as a function approximator to determine the next move. After all 50 games are played, the TD-learning algorithm chooses 500 different games randomly for each individual to learn. Exploration is not needed, because it learns from other games than its own games.

The NEAT-TD(λ) method is tested against the random opponent and against the three deterministic opponents. Both NEAT and TD(λ) have learned against these opponent, so this way the techniques can be compared.

8.1.1 Time

As explained in *chapter 4* the experiments with NEAT take a long time, from 8 to 10 hours for each run.
In the previous chapter an optimization was made resulting in a performance boost for NEAT so it takes about 2 - 4 hours for each experiment.
But because for NEAT-TD(λ) the TD(λ) technique has to be included and the number of actual games played is doubled (100 instead of 50 for each individual; see previous chapter) NEAT-TD(λ) also takes 8 hours for each experiment against a random opponent. The experiments against the deterministic opponents take even longer, up to 18 hours for each experiment against the mobility player. The mobility player is slow, because for every possible move it has to determine what moves are possible after making that move. So in fact it looks 1 step further than all other opponents.

Because these experiments take so much time, the number of different experiments are limited. More on this in the next chapters.

8.2 Random opponents

The Random opponent is the first opponent for NEAT-TD(λ). Both NEAT and TD(λ) were able to learn to defeat the random opponent. NEAT had a score of 92% in the end where TD(λ) scored 88% with a neural network of 10 hidden neurons and a value for λ 0.9 and for α 0.01.
As can be seen in *Figure 6.3* TD(λ) is able to learn a lot with a small amount of games reaching a score of 70% in maybe a few thousand games for all network sizes. That is a good thing, because to use TD(λ) with NEAT it cannot learn from 100.000 games or more; only a small amount of games is available for the TD(λ) in NEAT-TD(λ).
Because TD(λ) has a good score and is able to learn pretty quick the results of the next experiment will probably be at least as good as the results of the experiment with only NEAT and hopefully even slightly better.

First the amount of games needed for TD(λ) is tested. The TD(λ) part of NEAT-TD(λ) uses the 5000 games played each generation as a gamedatabase to learn from as described in the previous chapter. In this first experiment a number of 500 games and a number of 2000 games to learn from is tested. In this experiment the value for λ is 0.9 and the value for α is 0.01.

All experiments are repeated 10 times to ensure a good average.

The results of this experiment are shown in *figure 8.1*.

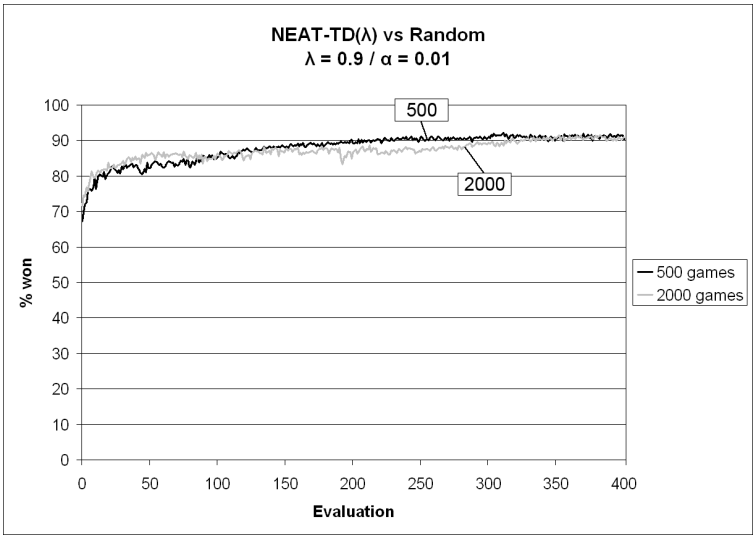

Figure 8.1: NEAT-TD(λ) vs Random with different number of games

First thing to notice is that there are not much differences between the results of 500 and 2000 games. Also, the score at the end of the 400 generations is around 92%, the same as NEAT itself.

So apparently the number of games the TD(λ) can learn from does not seem to matter that much. Worse, the TD(λ) does not seem to improve NEAT at all.
As for now, the number of 500 games is chosen over 2000 for the next experiments, because the experiments will cost less time this way.

When the results of NEAT, TD(λ) and NEAT-TD(λ) are joined together in one graph it is clear that NEAT and NEAT-TD(λ) do not really differ. This graph is shown in *figure 8.2*.

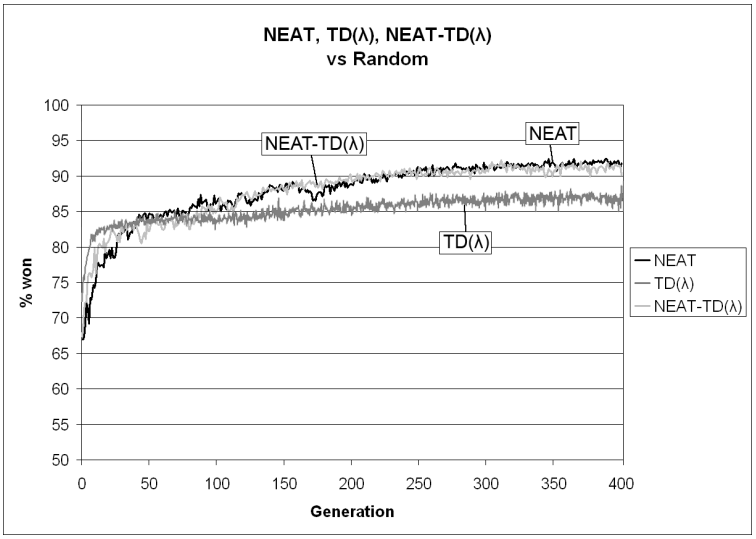

Figure 8.2: NEAT, TD(λ) and NEAT-TD(λ) vs Random

TD(λ) seems to learn quicker than NEAT and NEAT-TD(λ), but converges too early to a local optimum resulting in a 5% lower end score. NEAT and NEAT-TD(λ) have the same learning curve and end up with the same result, so for the random opponent, NEAT-TD(λ) does not seem to be better than just NEAT. Trying different parameters and performing other experiments might show if and how this can be improved. See also the following chapters.

8.3 Deterministic opponents

To see how well the NEAT-TD(λ) player performs against deterministic players, the player is tested against the TD-Greedy player, the Positional player and the Mobility player. These players are the same as used in previous experiments.
Different learning rates of the neural network are tested, because initial experiments showed that maybe a value of 0.01 for α is too high. Therefore also 0.001 is tested.
Because the NEAT-TD(λ) player and its opponent are both deterministic in these experiments, the first four moves in each game are played randomly resulting in 244 different initial states to ensure that different gamestates are visited during the game.

The TD(λ) part of these experiments use a selection of 500 games to learn from for each individual. A value of 0.9 for λ is used and 0.01 for α.

All experiments are repeated 10 times.

8.3.1 TD-Greedy opponent

First the NEAT-TD(λ) is tested against the TD-Greedy opponent, the deterministic player from the research in [Jac05].
NEAT was able to defeat this player 80% of the time after 400 generations and won half of the games after only 40 generations as shown in *figure 4.14*.
TD(λ) ended up with a 70% win percentage after 2.000.000 games with a very steep learning curve for the first 10.000 games. So TD(λ) might very well improve the NEAT method when the two techniques are combined or at least have the same result.

Both a learning rate α of 0.01 and 0.001 are tested. The outcome of this experiment is shown in *figure 8.3*.

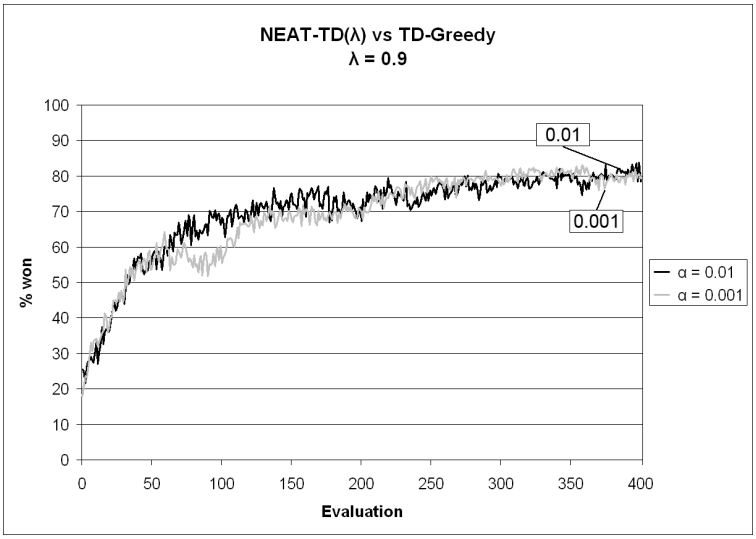

Figure 8.3: NEAT-TD(λ) vs TD-Greedy with different values for α

This graph shows that there is not much difference between the two values of α. Both showing a proper learning curve resulting in a win percentage of 80% after 400 generations. This is the same as NEAT alone. So it seems that NEAT-TD(λ) does not perform better than NEAT against the TD-Greedy opponent. This can clearly be seen in the graph in *figure 8.4* which shows the best results for NEAT, TD(λ) and NEAT-TD(λ).
Although it looks like the curve of NEAT-TD(λ) is just a little above the curve of NEAT, this is not a significant improvement. From generation 100 to generation 200, the curve of NEAT-TD(λ) is above the curve of NEAT, but this can be some bias and the difference in percentage is not significant.

In the next chapters more conclusions are drawn from these experiments.

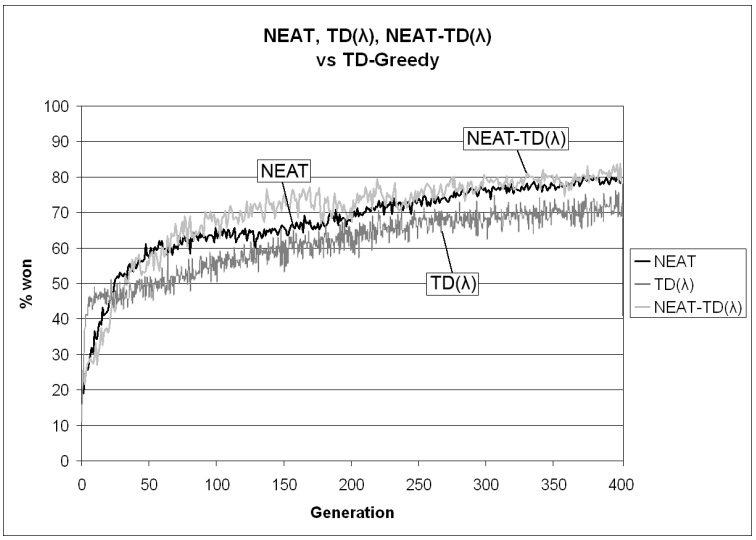

Figure 8.4: NEAT, TD(λ) and NEAT-TD(λ) vs TD-Greedy

8.3.2 Positional opponent

The next opponent is the Positional opponent. NEAT was able to defeat this player easily ending up with a 70% win percentage while still not converged.
TD(λ) on the other hand ended up with very poor results against the Positional opponent. It scored just over 40% and because of this low end result and the not very steep learning curve, the prognosis for the NEAT-TD(λ) experiment against the Positional player is not good. It is very well possible that TD(λ) will have a negative effect on the results of NEAT. It is not very likely that NEAT-TD(λ) will do better than NEAT alone.

Both a learning rate α of 0.01 and 0.001 are tested and the results are shown in *figure 8.5*. The graph shows a poor performance of NEAT-TD(λ) where α is 0.01 resulting in just a 50% win percentage. A value of 0.001 for α performs significantly better ending up with just about 62% after 400 generations.
But 62% is not nearly as good as the score of NEAT alone, 70%. TD(λ) itself had a low score and adding this technique to NEAT has a negative effect when learning against the Positional opponent. It is able to learn and it does win more than half of the games in the end, but the graph is almost converged so the results are worse than NEAT alone.
This can clearly be seen in the graph where all results of NEAT, TD(λ) and NEAT-TD(λ) are combined in *figure 8.6*.
NEAT has the best learning curve here and is still learning in the end.
More about this in the next chapters.

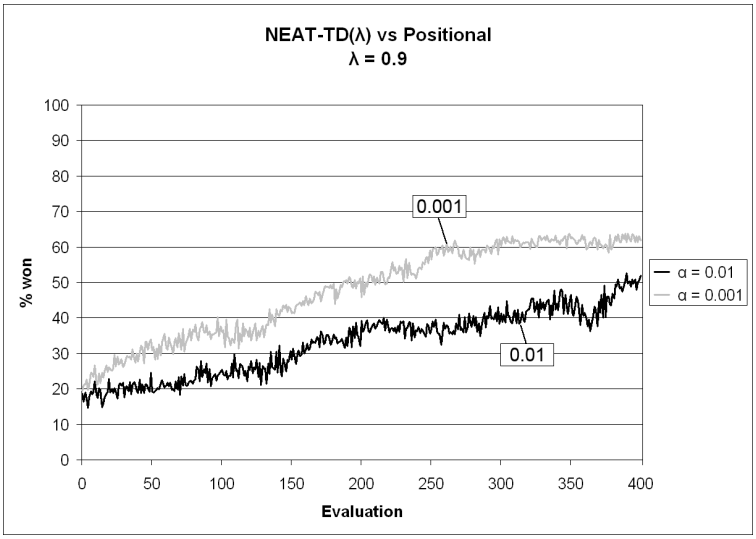

Figure 8.5: NEAT-TD(λ) vs Positional with different values for α

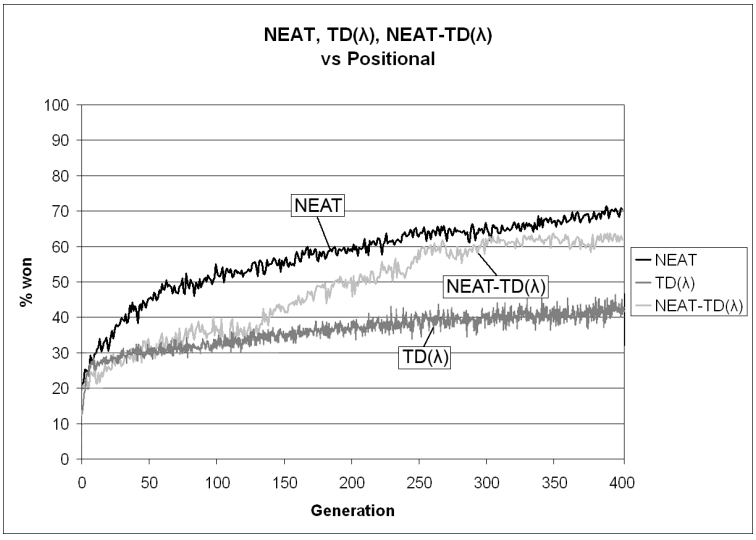

Figure 8.6: NEAT, TD(λ) and NEAT-TD(λ) vs Positional

8.3.3 Mobility opponent

The last deterministic opponent is the Mobility opponent. Both NEAT and TD(λ) were able to learn a good game against this opponent. NEAT had a very smooth learning curve and ended with a score of 77% while still not being converged after 400 generations.
TD(λ) had also not completely converged after 2.000.000 games and ended up with a score of 65%. It had a pretty steep learning curve in the beginning, which may have a positive influence on the combination of the two techniques.

To test NEAT-TD(λ) against the Mobility player, two values of α have been used, 0.01 and 0.001. The results of this experiment are shown in *figure 8.7*.

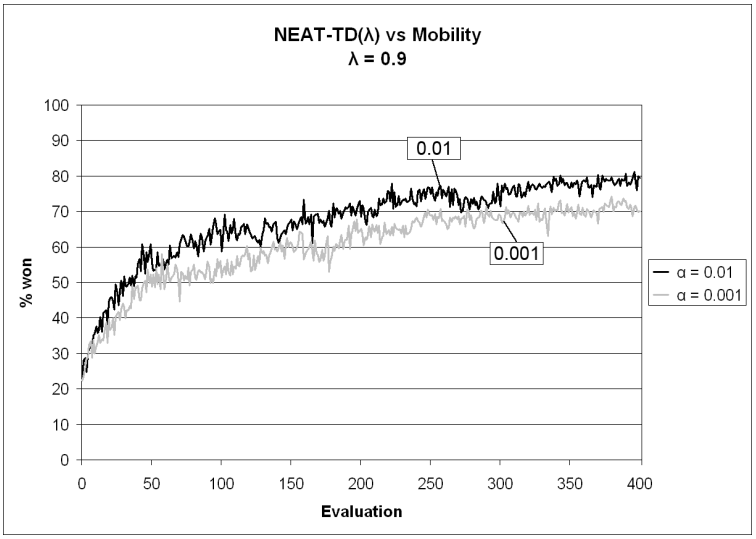

Figure 8.7: NEAT-TD(λ) vs Mobility with different values for α

It is clear that a value of 0.01 for α is significantly better than 0.001 resulting in a final score of 79%. This is even slightly better than NEAT itself. So the NEAT-TD(λ) combination does a good job against the Mobility opponent.
It shows a smooth learning curve and in *figure 8.8* where the graphs of NEAT, TD(λ) and NEAT-TD(λ) are joined together can be seen that NEAT-TD(λ) performs just a little better than NEAT.
This is not a significant improvement, but in almost all 400 generations the NEAT-TD(λ) scores about 2% higher than NEAT alone.

In the end, the results of NEAT-TD(λ) in the random experiments and all deterministic experiments are a little disappointing. The next chapters will go deeper into these results and draw conclusions from them.

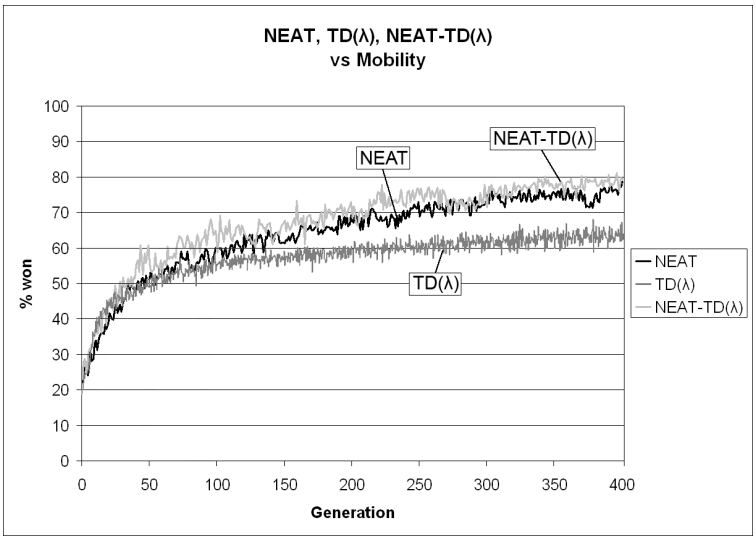

Figure 8.8: NEAT, TD(λ) and NEAT-TD(λ) vs Mobility

Chapter 9

Conclusion

9.1 Part I - Neuroevolution and Othello

Find the neuroevolution technique that is best at learning to play Othello.

Three neuroevolution techniques, SANE, ESP and NEAT, have been tested and compared in terms of learning potential and speed.
Looking at the results in *chapter 4* the first conclusion is that both ESP and NEAT perform much better than SANE. SANE ended with a 83% win percentage against the random opponent. That is not a bad score and several other researches with reinforcement learning and Othello show percentages like that. Probably with some optimizations SANE can perform even better. But it is expected that in the long run it will never be as good as ESP and NEAT.
ESP is a technique partly based on SANE, but more sophisticated in terms of specialization. ESP uses subpopulations to choose its neurons from which leads to faster specialization than SANE.
The results of ESP show a much better performance than SANE. There is a difference between ESP with and without delta coding, where the experiments without delta coding show better results. Maybe delta coding can be used to increase the end result, but further experiments are needed to find out. Without delta coding, ESP reaches a 88% win percentage against a random opponent which is significantly higher than SANE.
ESP shows good gaming potential. It has smooth learning curves and shows good progression over the generations. And because of its 40 hidden neurons it probably has good potential against different opponents because the last technique, NEAT, only uses a few hidden neurons to perform even better than ESP.
NEAT differs from SANE and ESP, because it evolves both weights and topology of the neural network. The results of NEAT are very promising. A steep learning curve in the beginning and a high win percentage at the end of the experiment. After 400 generations it had reached a win percentage of 92% against the random opponent and is still (slowly) learning. This is a better result than ESP and SANE, which gives NEAT good potential.
One remarkable observation was that the best networks only used 2 hidden neurons. The strategy includes a focus on only 2 corners of the board instead of 4. So apparently very few knowledge and a simple strategy is all that it takes to beat a random opponent.
To find out whether NEAT can perform against other opponents NEAT was tested against deterministic opponents. NEAT was able to learn a good game against all three deterministic opponents with scores varying from 70% to 80% and in all cases NEAT still had not converged after 400 generation. So NEAT shows very good potential as a method to learn to play Othello.
The final neural networks that NEAT had evolved against the deterministic opponents are

still very small. All networks had about 6 hidden neurons, varying from 4 to 8. This means that all these deterministic players can be defeated with a relatively simple strategy.

It is clear that NEAT has the best results. This is probably due to the fact that NEAT evolves the topology instead of only the weights, resulting in much more different neural networks. This makes NEAT a very powerful method because it can minimize the number of neurons needed resulting in a faster learning process. Both SANE and ESP use a fixed topology that has to be designed before the experiments. This results in bigger neural networks and they take much longer to learn.
Both ESP and SANE use a fixed topology and ESP is based on SANE, but is more sophisticated. So it is not a big surprise that it performs better.
NEAT and ESP are completely different, but both show smooth learning curves and good results. So which one has the best potential to learn an overall good strategy for Othello? Both have their strength and their weaknesses.
ESP uses a fixed topology which needs a good decision of the researcher before starting the experiments, but with a large network it is able to learn a good strategy. NEAT shows that a very small network is capable of defeating a random opponent, so ESP can probably get better and faster results with a smaller network against a random opponent, but that does not make it a good Othello player.
NEAT does not need a decision on its topology, but to be able to learn a good topology and strategy, it needs several different opponents, and who knows how big the networks will become when learning against several different opponents at the same time?
Because of time constraints it was not possible to do more experiments to find out how ESP and NEAT perform when learning against different opponents at the same time. New experiments are needed for that.

9.2 Part II - Neuroevolution and TD Learning

How can the neuroevolution technique NEAT be combined with TD(λ) to perform even better than the techniques on their own?

Because NEAT showed the best results in Part I, NEAT was chosen to be combined with the reinforcement technique TD(λ).
Before combining the two techniques TD(λ) was tested against all the available opponents. This is done because the research in [Jac05] was not sufficient to compare to. The implementation used in that research contained a bug, so the end results could not be used for this research. Also the new TD(λ) are better to compare to, because they also use 2.000.000 evaluations and uses the exact same opponents as the other experiments. The results of the TD(λ) experiments showed that TD(λ) is able to learn to defeat most of the opponents. Against the Random opponent it scores a 87% in the end, 70% was scored against the TD-Greedy deterministic opponent and a score of 62% was reached against the Mobility opponent. TD(λ) was not able to learn to defeat the Positional opponent. This is remarkable, because the Positional opponent uses the most simple strategy and can be defeated by a simple form of mobility strategy. Apparently the TD(λ) player did not visit enough different gamestates to learn a good strategy. Maybe more exploration is needed

to learn to defeat this player. Overall TD(λ) was able to learn to defeat its opponents, although the results are not as good as the results from NEAT.

NEAT and TD(λ) are combined in *chapter 7* to form a new technique NEAT-TD(λ). This technique should take advantage of both strong global search of NEAT and the high learning speed of TD(λ). NEAT extended with TD(λ) should perform better than NEAT itself. To test that, several experiments have been performed in *chapter 8*.

The results of these experiments are a little disappointing. NEAT-TD(λ) does not seem to be better than NEAT itself and sometimes performs even worse.
The combination of NEAT and TD(λ) uses the optimal parameters of both techniques. This means for TD(λ) it uses the parameters that are tested for a neural network size of 20 hidden neurons. And because NEAT starts with neural networks with 10 hidden neurons and ends up with small neural networks of 2 - 6 hidden neurons, TD(λ) should have been optimized for very small neural networks. Maybe this is one of the causes of these results.
Another problem is the number of games TD(λ) uses to learn from. Each individual could use 500 games to learn from, and maybe this is not enough to learn and improve the individual. A larger number of games, like 5000 or even more, is maybe better. The drawback of this is that there are only 5000 games available each generation. Another drawback is the time. More games take more time.
Because of time constraints the influence of the number of games could not be tested.

The most remarkable result was the result of the experiments against the Positional opponent. Earlier experiments showed that TD(λ) was not able to learn against this opponent and that NEAT was able to learn to defeat it. It is remarkable that the TD(λ) part of NEAT-TD(λ) has such a negative influence resulting in a significantly lower score than NEAT.
Apparently TD(λ) is not able to improve NEAT against some opponents, ending up with the same score as NEAT, and at the same time it can have a devastating influence on NEAT as well against a different opponent.
What is so different about the Positional opponent that NEAT-TD(λ) scores so much lower than NEAT against this opponent?

The biggest problem is that TD(λ) is not able to learn against this opponent. NEAT is able to learn against this opponent using a sophisticated global random search in the search space, as where TD(λ) cannot find a good solution at all and gets probably stuck in a local optimum very easily. Maybe different parameters for TD(λ) will solve this. A different exploration-exploitation ratio might help, or different values for λ. Because of time constraints this could not be tested.

In the end the results of NEAT-TD(λ) are a bit disappointing because they did not show an improvement compared to NEAT alone. Still it can be promising, but future research has to be done to find out how NEAT and TD(λ) can be combined better.

What can be said about learning Othello and learning board games in general?

NEAT and also NEAT-TD(λ) are able to learn to play a good game against several different opponents. But because all experiments used only one opponent at a time there is no information about how well these techniques will perform when learning against different opponents at the same time. Learning to defeat several opponents is learning a sophisticated strategy and that will probably need a larger neural network. And a larger neural network means that it takes longer to learn, so that is a whole new kind of problem. Another element of a good Othello player is looking ahead. In all experiments the player and its opponent only looked 1 step ahead. So only the next move was taken into account in making a decision. Learning to look ahead and learning against players who look ahead is also a whole new set of problems.

So in the end only a small part of the problem of learning to play Othello has been researched. The research shows potential, but because it is only a small part of the Othello problem, it cannot be used to say much about playing boardgames in general.

Chapter 10

Recommendations

10.1 Part I - Neuroevolution and Othello

Several improvements on the experiments could lead to an even better comparison of the three techniques.

To measure the performance of a technique 50 games of Othello were played by each neural network in the population each generation. At the end of the generation, the best neural network was allowed to play 1000 games to measure the champion's fitness of that generation. The problem here is that 50 games might be too few. There can be a lot of inaccurate results resulting in unjust champions, especially against a random opponent. So maybe more games are needed, or a different way of measuring the performance.

More interesting future research is learning against several different opponents. This way the result should be an overall good Othello player. Therefore more sophisticated Othello players are needed. The big disadvantage of this is that good opponents are slow which will slow down experiments dramatically.

Another interesting possibility is learning against itself. Have one technique play tournaments where neural networks play against each other every generation, and evolve the good ones. The advantage is that it is much faster than learning against slow sophisticated opponents, but the disadvantage is that no result is guaranteed because of the lack of known good opponents.

ESP showed good potential and can be optimized by experimenting with different population and subpopulation sizes and different delta coding values.

Other interesting areas are combining a good neuroevolution technique with other reinforcement learning methods.

Also optimizing the knowledge representation is an interesting area of research. Maybe put some more focus on learning strategies and use a priori knowledge of good strategies to speed up the learning process.

In this research only three techniques have been compared, while other techniques like TEAM or CoSyNe might have good gaming potential. It can be interesting to test these techniques as well.

10.2 Part II - Neuroevolution and TD Learning

The biggest problem in this research was the time it took to perform the experiments. Because of the time constraints not all possible parameters and situations were tested resulting in sub-optimal and maybe even bad results.

NEAT-TD(λ) performed badly against the Positional player. To improve results TD(λ) could be optimized to learn using small networks. This might also improve the results of the other experiments.

Also the number of games TD(λ) used to learn from may be too low. This can be part of new research as well. Probably a higher number of games to learn from will bring better results.
To optimize the speed, the implementation of NEAT-TD(λ) should be changed so that the TD(λ) part uses the games from the previous generation to learn from, instead of playing 50 games twice each generation. This will be a huge performance boost.

Overall several new experiments can be performed to tune the parameters of NEAT-TD(λ) so it will outperform NEAT. It still must be possible to use the power of TD(λ) to improve NEAT.

To find out how well NEAT and NEAT-TD(λ) perform against different opponents, the NEAT and NEAT-TD(λ) can be extended to learn against several different opponents at the same time, probably resulting in an overall better player.

Bibliography

[Ald02] M. Alden, A. van Kesteren and R. Miikkulainen. Eugenic Evolution Utilizing a Domain Model. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002), San Francisco, 2002, pages 279-286*

[And02] T. Andersen, K.O. Stanley and R. Miikkulainen. Neuro-Evolution Through Augmenting Topologies Applied To Evolving Neural Networks To Play Othello. *Department of Computer Sciences, University of Texas at Austin, 2002*

[And04] G. Andersson. Wzebra (Othello program). *http://www.radagast.se/othello, 2004*

[Bar02] M. Bardeen. TD-Learning and Coevolution: Hiding or Guiding. *Master thesis, University of Sussex, 2002,*

[Bil90] D. Billman and D. Shaman. Strategy knowledge and strategy change in skilled performance: A study of the game Othello. In *American Journal of Psychology 103, 1990, pages 145-166*

[Bur97] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. In *ICCA Journal 20(3), 1997, page 189*

[Del04] R. Delorme. Edax (Othello program). *http://abulmo.club.fr/edax/index.htm, 2004*

[Dri07] S. van den Dries. Het verkrijgen van evaluatiefuncties voor Othello met behulp van Temporal Difference Leren en Co-Evolutie. *Bachelor scriptie, University of Utrecht, 2007*

[Gho04] I. Ghory. Reinforcement Learning in board games. *Technical Report CSTR-04-004, Department of Computer Science, University of Bristol, 2004*

[Gom99] F.J. Gomez and R. Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI99, Stockholm), Denver:Morgan Kaufmann, 1999*

[Gom06] F.J. Gomez, J. Schmidhuber, J and R. Miikkulainen. Efficient Non-Linear Control through Neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML-06, Berlin), 2006*

[Gro65] A.D. DeGroot. Thought and Choice in Chess. *Mouton, 1965*

[Hsu04] Feng-Hsiung Hsu. Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. *Princeton University Press, 2004*

[Jac05] S.M. Jacobs, T. Mioch, W. Tinus and M. Vuurboom. Envy, Greed and Pride - The three sins of RL Reversi. *Project report, University of Utrecht, 2005*

[Lub01] A. Lubberts and R. Miikkulainen. Co-Evolving a Go-Playing Neural Network. In *Genetic and Evolutionary Computation Conference Workshop, GECCO 2001, San Francisco, 2001, pages 14-19*

[Meg04] F. Di Meglio. OthBase (Othello game database). *http://www.othbase.net, 2004*

[Mor95] D.E. Moriarty and R. Miikkulainen. Discovering Complex Othello Strategies Through Evolutionary Neural Networks. In *Connection Science 7(3), 1995, pages 195-209*

[Mor96] D.E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. In *Machine Learning 22, 1996, pages 11-32*

[Mor97] D.E. Moriarty and R. Miikkulainen. Forming Neural Networks Through Efficient and Adaptive Coevolution. In *Evolutionary Computation 5(4), 1997, pages 373-399*

[Per01] A.S. Perez-Bergquist. Applying ESP and Region Specialists to Neuro-Evolution for Go. *Honors Thesis, Department of Computer Sciences, University of Texas, Technical Report CSTR01-24, 2001*

[Pri98] J.W. Prior. Eugenic Evolution for Combinatorial Optimization. *Master thesis, Department of Computer Sciences, University of Texas, Technical Report AI98-268, 1998*

[Rus95] S.J. Russel and P. Norvig. Artificial Intelligence - a modern approach. *Prentice-Hall International Inc, 1995*

[Sta02] K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. In *Evolutionary Computation 10, 2002, pages 99-127*,

[Ste96] C. Stergiou and D. Siganos. Neural Networks. In *SURPRISE 96 Journal, http://www.doc.ic.ac.uk/ñd/ surprise_96/journal/vol4/cs11/report.html, Imperial College of Science Technology and Medicine London, 1996*

[Sut88] R.S. Sutton. Learning to Predict by the Methods of Temporal Differences. In *Machine Learning 3, 1988, pages 9-44*

[Sut98] R.S. Sutton and A.G. Barto. Reinforcement Learning: An Introduction. *MIT Press, 1998*

[Vel99] A. Vellidoa, P.J.G. Lisboaa and J. Vaughan. Neural networks in business: a survey of applications. In *Expert Systems with Applications 17, 1999, pages 51-70*

[Whi06] S. Whiteson and P. Stone. Evolutionary Function Approximation for Reinforcement Learning. *Journal of Machine Learning Research 2006*