Playing *GO* with an Artificial Neural Net Simulater

For the subject *An introduction to biological computer science*

Nicolas Goell
nic-goell@magnet.at

Roland Knall
roland.knall@gmx.net

Mark Rehbichler
mark-rehbichler@magnet.at

DI Mag. Roland Schwaiger[1]
rschwaig@cosy.sbg.ac.at

Version: 5th of july Time: 1:00pm

[1]project coordinator and supervisor

Contents

List of Figures

Abstract

GO is a possible application for *Artificial Neural Networks*. We discuss the possibilities of playing GO and ways to implement it.

Chapter 1

Introduction

In this chapter we will give a short overview about GO, and what kind of work has already been done, by previous classes. Basically we will give a review of [1].

1.1 Playing GO - Rules

Although the word GO is taken from the Japanese word for the game, it seems to be coming from China about 3000 years ago. It is a simple board based strategy game, but it appears simple in comparison to chess or similair board based strategy games. The main goal is to reach more points than your opponennt. To do so, you can place stones, black or white, on a square grid with 19 intersections across. Two players, black and white, place alternate a stone of their respective colors on an intersection. The games end when both players pass or no more space for placing stones is left. Compared to chess, there exists also a system of experience classification. If a player has more experience, he allows his oppenent a number of free moves. In that case it is said that a player ranks x stones higher. (x is the number of allowed free stones)

1.1.1 How to gain points

Each player tries to catch stones of the opponent. To do so, the player has to place them around the opponent stones. If stones got "captured" in such a way, they are removed from the board. At the end of the game the remaining stones are counted. If both players have the same skill, white is allowed to add a number of points (usually 5.5), offsetting the fact of black's first move. The player with the highest score wins.

1.1.2 Parts of the game

A game of GO can be divided into three general phases: the opening, the midgame and the endgame. Computer's success in each of this stages varies and we have to take a deeper look in what can and what can not be achieved with computational methods.

The Opening

The opening game is referred to as the fuseki. Specified combinations of stones in the beginning are referred to as joseki. Good GO programms contain databases with 5000 to 50000 moves, and they have no difficulties to play a good beginning with them. But because evolving a good strategy is still a bigger goal, improvements were not made yet in this stage of the game.

The Midgame

This stage of a game contains a greater challenge for computer programs. Because of the sheer size of a average board (19x19), between 200 to 300 moves are possible on a first sight in a typical midgame position. Therefore brute force searching would result in much too larger trees. Techniques which are used contain pattern libraries, tactical analysis and rule-based expert systems. The system first evaluates candidate moves, using one or some of the named techniques and analysis them.
There are several problems with the current midgame techniques. They tend to be difficult to apply and error-prone, and because most of them are static, it is difficult to achieve general play and advance must be made by a human.

The Endgame

The endgame of GO is more simpler for computer programs, but usually there are less points to accumulate. The board is full, patterns are set and positions are taken.
Although computer programs play very well in this stage, they achieve less points compared to what could be taken out of the midgame. So the most important and difficult part of playing GO is in the midgame where the traditional techniques are the weakest.

1.2 Neural Networks and GO

For a computer GO is really hard to learn, and the best networks are still weaker as an average human player. There exists a number of reasons for this, just too name a few:

1. The games is pattern oriented.
2. The branching factor is prohibitively large, what means that a possible move may lead to a large number of possible next moves, which increases calculation time.
3. There are also multiple interacting goals.

Probably evolving new methods and networks can solve these (and many other problems connected to this subject) problems. But the main factor, that is holding back computer programs, is the largely pattern oriented game strategies that need to be incorporated for successfully playing GO. As *Artificial Neural Network*s are known to be exceptionally good at pattern transformation tasks, employing this relatively new technique seems like a very promising approach.

1.2.1 SANE

SANE (*S*ymbiotic *A*daptive *N*euro *E*volution) [**?**] is a specific approach of an evolutionary *Artificial Neural Network* (further refered with *ANN*) not depend on backpropagation as training method which cannot be effeciently applied on the intended purpose. It maintains a large pool of neurons, structured as a classic three-layer feedforward network (see glossary on page 17), and additionally a population of blueprints which can be thought of as a selection of neurons specifying an individual neural network. This ensures the evolution of a wide variety of assemblages to compensate for the rather long evaluation intervals. A neuron's activation is defined by the transfer function $\sigma(x) = \frac{1}{1+e^{-x}}$. During the process of evolution SANE evaluates each blueprint's performance and calculates a fitness value for every neuron in the pool based on the performance of the networks in which it participates. In the so called reproduction phase, the top 25% in the neuron pool and top 15% in the blueprint population are randomly mated making use of a one-point crossover procedure to obtain new individuals replacing the least fittest ones. On the blueprint level, each link to a breeding neuron is altered to its offspring with a 50% probability. Additionally a conservative mutaion rate of 1% is applied to both populations.

1.2.2 Applying SANE to Go

SANE evolves common three-layer feed-forward networks with input layers of two units (Boolean value) and output layers of only one unit (float value) per board position. Positive output results indicate good and negative results bad moves. The neurones generated are 312 bits long and represent a set of 12 weights. Each generation 200 networks were formed with a mutation rate of 0.1% and 25-30% of the population allowed to breed, which has given good results during experimentation. It showed to be rather difficult to determine the end of a game, to remove dead stones and to check for repetition of entire board positions. Solutions for all these problems were found. An upper bound was placed on the number of moves in order to ban excessively long games. The fitness level for the networks was produced by summing the difference in score between SANE and its opponent over a certain number of games.

1.2.3 Results

SANE was tested with board sizes of 5x5, 7x7 and 9x9, where it was able to beat its opponent program wally after an acceptable number of generations. Larger boards could not be tested due to lack of CPU power. Several experiments showed, that SANE managed to beat deterministic opponents very rapidly, whereas it did not learn to play against more randomly acting players. When trained with a more non-deterministic wally the network developed tricks instead of only being able to win against only one certain player. In order to train the networks at a higher level the GO handicapping mechanism was used to simulate stronger opponents.

1.2.4 Strategies Evolved

An interesting question is, what kind of strategies did the networks evolve. Originally they started with no prior knowledge on how to play the game. Important for the network is the training partner, who is the only source of information for the network. A network with only one specific training partner can work out strategies to take advantage of the opponents weakness. For that reason it is better to use a variety of opponents. The networks outputs are random at the beginning but after a few generations they start to make simple living groups. The evolution system discovered good GO playing strategies without prior knowledge entirely on its own, based on what moves led to wins and losses

1.2.5 Future Work

Future Work will focus on scaling up to larger boards. It is planned to modify the program to be able to play on board size because now the networks can only play on the board on which they were developed. There are three possibilities to solve this problem:

1. to design a representation that is independent of board size
2. to work out solutions that only consider a portion of the board at a time, but by this method the result is a weaker play.
3. enhancing the network architecture.

There are several ways to make the architecture more effective, including pre-programmed feature detectors and hierarchies of networks. The networks are forced to discover useful features themselves because they are not given any prior knowledge about what features are relevant to playing GO. A pre-defined feature space instead of looking at the raw board might make the assignment easier and might allow the network to learn faster and deal with more complex patterns. and evolving against stronger opponents. A variety of stronger opponents is necessary to achieve truly high levels of play. The stronger player can be simulated by using handicaps but handicap games alone are not enough. A problem is that strong opponents tend to take longer to generate moves than weaker programs. In future it would be necessary to work out networks against human players. This way perhaps GO-playing could finally be evolved that were able to compete with the best humans.
New techniques based on pattern recognition and learning like SANE will be required to build GO programs that would be competitive at high levels of play because the traditional methods have been insufficient.

1.3 Former Projects

We want to make some conclusions concerning relative projects. During preparation for our work, we took a look on given papers and programs. During this collection of facts, we found some points we have to criticize and others we find worth to be brought in, to our work.
A first paper was already summarized in the introduction, so we will give only some short statements for the next one.

1.3.1 *Neural Networks playing GO* [2]

This group used for their work (as made clear in [2]), SNNS and one-hidden layer feed forward nets. On one side, they only used standard ANNs (in further shorted with ANN), so they did not implemented an evolutionary aspect, which improves, in general, the output of the program (see 1.2.1).

1.3.2 Improvements over the given work

In [1] and [2], we found some aspects, we would improve in our work.

The number of input units In [2] the number of input units was far too large, but this problem occured also in [1]. With a better coding of the given board situation, the number of input nodes could be reduced, and so the complexity of the *ANN*.

Hidden Layers Improvements can be made by using more layers in combination with more neurons. Also a feedback between neurons in hidden layers, could result in better performance. Both dicussed papers only used hidden layer, with many ([1]) or, with respect to the complexity of the subject too few ([2]) nodes in the hidden layer.

The hidden layer was fully connected to both the input and the output layer. One paper ([1]) used SANE for keeping the number of connections low. The other ([2]) connected every node in the hidden layer with each of both the input and the output layer. Using a evolution algorithm such as SANE (see 1.2.1), the number of connections between input and output layers and the hidden layer structure can be optimized.

Number of training patterns Strategics used for GO are, on the one side pattern oriented, on the other side rule based. Therefore to train an *ANN* with prepaired games, a large number of training patterns should be used, i.e. a huge number of games. But that costs training time. Therefore we will have to evaluate the best combination of training patterns and computation time.

Chapter 2

Our work

2.1 Coding an input layer structure

In our given papers the simplest form of coding could be found. Every intersection could be found in the input layer, which increased the number of nodes in the input sets. Therefore we wanted to find better possibilities for coding the input layer. Also we want to find out if this takes affect in the net performance. We do not include the simplest form of coding in the final net, but we will use it as reference later.

The simplest form of coding would just take every intersection, calculate a value for it, and give the whole set to the net. The value could be everything, what identicates correctly the different possibilities (emtpy, black, white) for an intersection.

2.1.1 First approach (Standard)

Divide the board into a number of pieces of equal size. For example a 9x9 board could be split up into nine parts of 3x3 or 27 parts of 3x1.
For each board sector a value is calculated depending on where black or white stones can be found. E.g. in a 3x1 sector there are $3^3 = 27$ possibilities of placing stones. Set EMPTY = 0, BLACK = 1, WHITE = 2 and calculate the value by converting the resulting base-three-number into decimal system. In our 3x1 sector [EMPTY, WHITE, WHITE] means $220_3 = 24_{10}$.
As can be seen, larger sectors may be inefficient, as the neural network does assume completely different patterns to be quite similar. So in a 3x3 there are $3^9 = 19683$ possible patterns. Assuming that the lower right corner of the sector is valued by 3^0, the patterns value in decimal system varies only by a maximum of two - no matter if the position in question is EMPTY, BLACK or WHITE.

Example:

```
Pattern 1

[BLACK,BLACK,WHITE,
 EMPTY,WHITE,WHITE,
```

```
 BLACK,EMPTY,WHITE]
```

is transfered to $112022102_3 = 10433_{10}$

```
Pattern 2

[BLACK,BLACK,WHITE,
 EMPTY,WHITE,WHITE,
 BLACK,EMPTY,BLACK]
```

is transfered to $112022101_3 = 10432_{10}$

As can be seen easily, two quite different patterns result in nearly equal decimal values. The *ANN* will hardly be able to distinguish between the two samples.

2.1.2 Second approach (Influence)

In order to develop a GO program based on *ANN*s that plays at a reasonable level, it seems necessary to provide additional data to the input layer, than simply supplying the position of each stone on the board. Therefore each piece could be assigned a sphere of influence. Instead of just marking the particular board position with either 1 or -1 (depending on the player) , a higher value like +/-50 would be used and radiated to surrounding fields in a decreasing manner. The representation of a single white stone at position c3 is shown in picture 2.1.2 on page 8 If black moves to b2, we get the situation shown in picture 2.1.2 on page 8 Now let's suppose white places its stone at c2, this would result in the situation shown in picture 2.1.2 on page 9
You can see that the influence and probable territory in the second situation is well balanced, but as soon as white moves to c2 to threaten the opponent's stone, black's control is driven back to the upper left corner. As a result the net should be able to extract more information from a given game situation.
This method was first outlined 1971 by Jon Ryder, though as basis for an expert system rather than a neural network.
Another issue of evolving a GO playing *ANN* is the bad scalability of this approach to larger boards such as 19x19. To circumvent this problem, small groups of 2x2 or 3x3 fields could be represented by a weighted value. This scaled down version of the board would be presented to an evolved network, which should then indicate which group yields the greatest potential for a good move. Each field in the proposed area is further investigated by a second, standard feedforward net that is specialised on guessing which move to a specific position should be preferred, considering only the near environment. In contrast to the first network, the input units are fed with a subset of the game board, for example 5x5 fields with the center being the position that is to be investigated, but therefore in full detail. The input is then projected onto only one output unit indicating the strength of this particular move. Training could be accomplished with a set of recorded GO matches from professional players by a backpropagation algorithm.
Using a 2x2 field subdivision and a weighting function that simply calculates the mean value of each group could give the representation in picture 2.1.2 on page 9 containing the

situation shown in picture 2.1.2 on page 8 If the first net suggested a move to quadrant B1, the fields c1, d1, c2, d2 and their surroundings are subsequently presented to the second network for evaluation. In case position c2 is being considered as in picture 2.1.2 on page 10 The fields marked “OB” in the above illustration denote that they are off the board. This procedure yields four values, from which the highest will be picked for the next move (ideally this should be the one representing c2).

	a	b	c	d	e	f	g	h	j	
1										1
2										2
3			O							3
4										4
5										5
6										6
7										7
8										8
9										9
	a	b	c	d	e	f	g	h	j	

	a	b	c	d	e	f	g	h	j	
1	0	0	7	0	0	0	0	0	0	1
2	0	10	20	10	0	0	0	0	0	2
3	7	20	50	20	7	0	0	0	0	3
4	0	10	20	10	0	0	0	0	0	4
5	0	0	7	0	0	0	0	0	0	5
6	0	0	0	0	0	0	0	0	0	6
7	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	0	0	0	0	8
9	0	0	0	0	0	0	0	0	0	9
	a	b	c	d	e	f	g	h	j	

Figure 2.1: Representation of a single white stone

	a	b	c	d	e	f	g	h	j	
1										1
2		X								2
3			O							3
4										4
5										5
6										6
7										7
8										8
9										9
	a	b	c	d	e	f	g	h	j	

	a	b	c	d	e	f	g	h	j	
1	-10	-20	-3	0	0	0	0	0	0	1
2	-20	-40	0	3	0	0	0	0	0	2
3	-3	0	40	20	7	0	0	0	0	3
4	0	3	20	10	0	0	0	0	0	4
5	0	0	7	0	0	0	0	0	0	5
6	0	0	0	0	0	0	0	0	0	6
7	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	0	0	0	0	8
9	0	0	0	0	0	0	0	0	0	9
	a	b	c	d	e	f	g	h	j	

Figure 2.2: Black stone set to b2

2.1.3 Third approach (Standard - Extended)

This is a similar approach as standard encoding with the difference being the fact that only 0 and 1 are valid input values and the game situation is encoded seperately for black

	a	b	c	d	e	f	g	h	j	
1										1
2		X	O							2
3			O							3
4										4
5										5
6										6
7										7
8										8
9										9
	a	b	c	d	e	f	g	h	j	

	a	b	c	d	e	f	g	h	j	
1	-10	-10	17	10	0	0	0	0	0	1
2	-13	-20	50	23	7	0	0	0	0	2
3	-3	10	60	30	7	0	0	0	0	3
4	0	3	20	10	0	0	0	0	0	4
5	0	0	7	0	0	0	0	0	0	5
6	0	0	0	0	0	0	0	0	0	6
7	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	0	0	0	0	8
9	0	0	0	0	0	0	0	0	0	9
	a	b	c	d	e	f	g	h	j	

Figure 2.3: White stone set to c2

	A	B	C	D	E	
1	-22.5	0	0	0	0	1
2	0	22.5	1.75	0	0	2
3	0	1.75	0	0	0	3
4	0	0	0	0	0	4
5	0	0	0	0	0	5
	A	B	C	D	E	

Figure 2.4: Grouping the board situation

and white stones. For consequence the is input layer twice as large compared to approach one (see 2.1.1) and two (see 2.1.2).

2.1.4 Fourth approach (Prefiltering the board)

We wanted to make an approach, which allows the *ANN* to play in a more aggressive way. Therefore the board gets divided up into four sections of the same size. As information for the input layer we allow only the section, which contains the las opponent move.
This approach can also be used for filtering the board information for another *ANN*. So you can reduce big sized boards to small ones, which decreases calculation time. Also a better pattern recognition should be possible. Because of a lack of time we were not able to test this approach with a *ANN*.

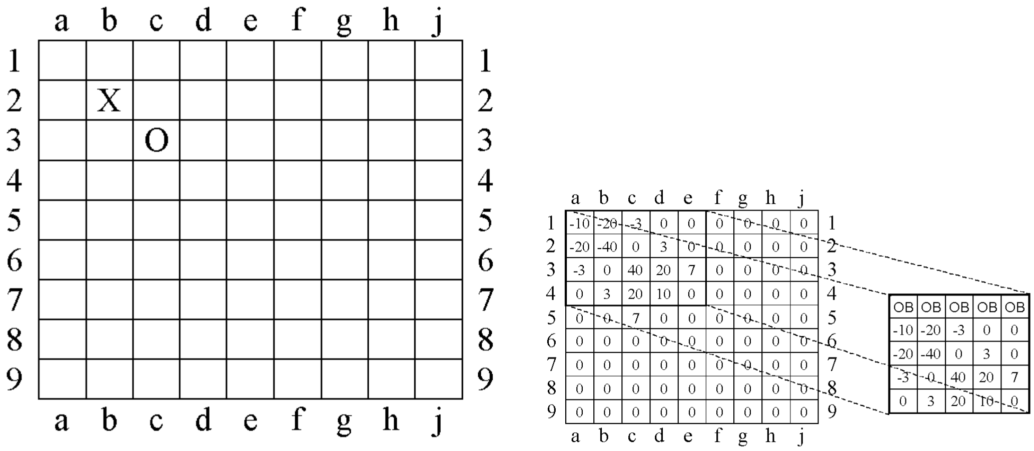

Figure 2.5: c2 is being considered for move

2.2 First tests using our codings

After discussing various possibilities for coding an input layer structure, we started first tests, to see how good (or bad) our approaches will work.

2.2.1 First approach

This version of coding the board was tested using two different feedforward networks. The first worked with one hidden-layer of 300 neurons, the other one had a hidden-layer with 1000 neurons. Both used 27 input-items and 19 output-items. Weights were initially randomized. As the learning function a standard backpropagation (see Glossary on page 17) algorithm was chosen. As SNNS offers only the possibility of full-connection (alternatively connections can be established manually) feedback-connections or similar working models were not implemented. The training-file consisted of 50 training-patterns, where each represented a specific game situation in GO. The smaller network passed between 2000 and 10000 training-cycles, whereas the larger one - due to lack of time and computation power - was stopped after about 1000 cycles. The results of these initial tests were not very promising. The network with only 300 hidden-units did hardly manage to memorize the 50 given training-patterns. Even after 10000 training-cycles SSE (a value for the standard error) was between 58 and 63. The analyse of the resulting output-patterns showed, that only about 20% of the given moves were correctly memorized.

Mostly the results differed significantly from the original patterns. Using the larger network interestingly hardly improved the learning-process. SSE was slightly better, but it did not drop below 50. Because of the bad results using 50 training-patterns further tests used only 25 or ten patterns. Again the networks trained between 2000 and 10000 cycles. Although the number of patterns to learn was definitely smaller, the results were not significantly better than before. Finally a network with only 50 hidden-units was tested. Surprisingly SSE dropped below 10 (using 50 training-patterns), and the net was able to memorize 70% and more of the offered patterns. Obviously fewer hidden-units give better results than a large number. Doubtlessly it is quite important to adapt the number of hidden-units to the specific problem.

2.2.2 Second approach

The above table show the different network's ability to correctly reproduce the suggested move for a given input pattern. The standard map uses a simple encoding technique, where black, empty and white fields are represented by the numbers 1, 0, -1. In the influence map each stone radiates distance dependent influence values to sourrounding points (as described in the former paragraph). Therefore the data contained encodes the same information, but provides a richer input space for a specific game situation. However, the results show that the neural nets have difficulties in extracting this knowledge or rather take longer to adapt their weights and biasses. The simpler encoded maps are learned much faster, with higher accuracy and less neurons. It may well be the case that the nets trained with patterns from the influence map are more versatile recognizing similar game situations, but this is only a guess and would require further research to confirm.

Additionally the table reveals the importance of choosing the right number of hidden neurons for successful learning. Too little of them mean that the network cannot store enough information in order to exactly reproduce the output patterns. If too many neurons are present, the neural net's efficency decreases, although all networks are trained the same number of cycles. These nets may have the potential to collect and store a great quantity of information, but it takes a large amount of time to train them.

In an experiment the above trained *ANN*s challenged wally. Although they showed good performance in regard to reproducing the training patterns, none of the *ANN*s had a chance of beating the expert system, because a set of approximately 600 patterns on a 9x9 board is far too little to properly reflect the huge amount of possible game situations. Therefore it seems advisable to incorporate this kind of networks for solving problems, whoose evaluation requires only a small subset of the entire board.

2.3 Final results

In the last chapter we gave simple possibilities to enhance a net for palying GO. We created some simple net and started training them.

	Standard	**Standard-Extended**	**Influence**
25 hidden	184	123	930
50 hidden	160	98	846
150 hidden	162	86	717
300 hidden	320	156	555

Values in the table above are SSE after 1000 cycles of training using a momentum backpropagation technique with a training set of 576 patterns and a full connected hidden layer

(see glossary on page 17 for details).

Further investigation of the *ANN*s (this time including a third encoding scheme) in respect to their ability to learn a great variety of different game situations indicate that the nets benefit from a simple encoding technique and are scaling well regarding the large training set. Looking at the results it is evident that standard encoding and the extended version of it are most efficient. Taking into account that standard encoding manages to do with only half the number of input neurons, it seems reasonable to stick to it, though the extended variant produces lower error rates.
Considering the hidden layer's size, an amount of about 150 neurons appears to be functional.

Appendix A

Report of Meetings

1st Meeting

Date: 13th March 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Knall, Roland Schwaiger

Discussion: Our assisting project controller came up with some past work on our subject. He came up with a paper, which will be included in this one, in a shorter version (see 1). Also we defined a modus operandi for the term, discussing the sections, in which this paper will be divided. We also talked about the basic materials for this seminar, which can be found online.

2st Meeting

Date: 20th March 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Knall, Brigitte Kendlbacher, Roland Schwaiger

Discussion: We discussed the paper, we had to summarize. We talked about the pro's and con's of solving the problem with SANE, and what SANE in general is. We also sorted out some problems which could be improved during this work.

- Only one hidden layer is used.
- Nodes in the hidden layer can only connect to nodes from either the input or output layer. No information flows between nodes in the hidden layer.
- The number of neurons in the input and output layer is big.

We also dicussed solutions for this problem, including the following:

- Changing the possibilities for nodes to connect to other nodes.

- Evaluation of the "blueprints"
- General changes could be done to the input and output layer.

We also talked about the realization of *ANN*s with NetJen. And the differents between the solution using SANE and a solution using NetJen.

3th Meeting

Date: 27th March 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Knall, Brigitte Kendlbacher, Roland Schwaiger

Discussion: Beside representing the paper so far, we talked about JaGo and possibilities for us to realize a version of our project. Also we set the work which should be done in the next two weeks.

4th Meeting

Date: 11th April 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Schwaiger

Discussion: The paper so far, and each coding possibility got discussed. As work for the next two weeks we should declare the codings more exactly, and get in touch with the *SNNS* [3]

5th Meeting

Date: 2nd May 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Schwaiger

Discussion: We went into deeper discussions about our work so far. Especially the various parts of input codings had been a subject to discuse. For the next three weeks we are going to test each part of coding on its ability to be used in this project.

6th Meeting

Date: 23th May 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Knall, Roland Schwaiger

Discussion: The tests brought up some surprisingly results (see **??**), we reformed one approach to a filter idea for a net.

7th Meeting

Date: 6th June 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Schwaiger

Discussion: Again the tests were discussed, but also we made approaches of using SANE and Wally, especially we came up of the idea, let the wally play against our evolved net.

8th Meeting

Date: 20th June 2000

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Roland Knall, Roland Schwaiger

Discussion: Mainly setted work for the last few days, and talked about the presentation.

Bibliography

[1] N. Richards, D. E. Moriarty, and R. Miikulainen, *Evolving Neural Networks to Play GO.* University of Texas, Austin, 1997.

[2] G. Haberl, P. Maier, B. Neuhofer, and S. Sodar, *Neurale Netze spielen GO.* University of Salzburg, Institute for Computer Science, 1998.

[3] U. Stuttgart, *Stuttgarter Neuronal Networks Simulator.* 1995.

Appendix B

Glossar

- **backpropagation** Backpropagation is a learning algorithm. After running a training set, the algorithm adepts the weights in that kind of way that nodes which are near the result get higher weights. *First explained in chapter* 16
- **branching factor** A program for move bassed games such as GO or chess often calculates further moves. That way it examines chances for the next move. The deeper the branch goes, the longer calculation takes. *First explained in chapter* 16
- **full connected layer** A layer is called full connected, if every node from itself is connected to every node of his surrounding layers. *First explained in chapter* 16
- **hidden layer** The layer structures between the input and the output layer are called hidden layer. *First explained in chapter* 16
- **three layer network** This nettype consists of an input, hidden and an output layer, which may be fully connected. Information processes only in one direction, from the input, over the hidden to the output layer. *First explained in chapter* 16