Playing *GO* with a Artificial Neural Net Simulater

For the subject *An introduction to biological computer science*

Nicolas Goell
nic-goell@magnet.at

Brigitte Kendlbacher
brigitte.kendlbacher@gmx.at

Roland Knall
roland.knall@gmx.net

Mark Rehbichler
mark-rehbichler@magnet.at

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

Version: 19th of march   Uhrzeit: 04:00pm

[1]project coordinator and supervisor

Contents

Abstract

GO is a common used application for *Artificial Neural Networks*. This paper takes a look on the theoretical side, and shows up a realization using **netJEN**.

Chapter 1

Introduction

In this chapter we will give a short overview about GO, and what kind of work has already be done. It exists in main parts as a summary of [1].

1.1 Playing GO - Basics

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: the games is pattern oriented, the branching factor is prohibitively large and there are multiple interacting goals. Probably evolving new methods and networks can solve these (and many other problems connected to this subject) problems.

1.1.1 GO, the original game

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 games, but is appears simple in comparison to chess or similair board-strategy games. The main goal is to reach more points than your opponennt. To do so, you can playe stones, black or white, on a square rid with 19 intersections across. Two players, black and white, place alternate on stone of their respective colors on an intersection. The games end when both players pass or no more space for placing stones is left. It is very easy to even the opponent levels. If a player has more experience, he allows his oppenent a number of free moves. You say he ranks x stones higher. (x is the number of allowed free stones)

How to gain points

Each player tries to catch stones of the opponent. To do so, he has to place them around the opponent stones. If stones got "captured" in such way, they are removed from the board. At the end of the game the remaining stones are counted. In a normal even game, white can add 5.5 points, offsetting the fact of black's first move. The player with the highest score wins.

1.1.2 Computers and GO

Although the rule set is relatively simple, computers are still not able to play against good players and win. 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 difficults 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 the bigger problems for computer. 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 big 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 simplier for computers, but normally there are less points to achieve. The board is full, patterns are set and positions taken. Although computers 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

The main factor, that is holding back computer GO programs, is the largely pattern oriented game strategies that need to be incorporated for successfully playing GO. As neural networks 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 (Symbiotic Adaptive Neuro-Evolution) is a specific form of neural network that does not depend on backpropagation as training method which cannot be effectively applied on the intended purpose. It maintains a large pool of neurons, structured as a classic three-layer feedforward network, 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 because they start 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. In the future they plan to play on any board size because now the networks can only play on the board on which they were developed. There are two 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 preprogrammed 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 Project: *Neural Networks playing GO* [2]

We want to take some conlusions over the given work. 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 want to take over.
A first paper was already summarized in the introduction, so we will give only some short statements for the next one.

1.3.1 Discussing the paper

This group used for their work (see [2]), SNNS and one-hidden layer feed forward nets. One the one side, they only used standard *Artificial Neural Network*s (in further shorted with ANN), so they did not implemented an evolution, which improves, in general, the output of the program (see 1.2.1).

- The number of input units was far too large for the given situation.
- The number of neurons in the hidden layer seemed to be less than necessary.
- The hidden layer was fully connected to both the input and the output layer.

- The use of only one hidden layer complicates implementations of feedback algorithms.
- A variable system, which can be evolved, is more able to adapt the complex strategies of GO.
- Go is far too complex. So you should use more than the given training patterns.

1.4 Improvements over the given papers

We also found some things, we would improve in further work. This relates not only to [1] but also to the named paper [2] above.

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 net.

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 used only on hidden layer, with many ([1]) or, relied 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 I/O 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 tricky. Therefore to train a *ANN*s with games, you should use a large number of training patterns. But you have to keep in mind, that training costs time. Therefore you will have to evaluate the best combination of training patterns and computation time.

Chapter 2

Creating a Neural Network for playing GO

After discussing work which already has been done, we will go on creating a network for playing GO. We will start coding an input layer, go on defining rules for hidden layers, and an output layer.

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 blew up 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 possibility

Divide the board into a number of pieces of equal size. E.g. 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 believes 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]
```

means $112022102_3 = 10433_{10}$

```
Pattern 2

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

means $112022101_3 = 10432_{10}$

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

2.1.2 Second possibility

In order to develop a go program based on neural networks 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 If black moves to b2, we get the situation shown in picture 2.1.2 Now let's suppose white places its stone at c2, this would result in the situation shown in picture 2.1.2

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 neural network 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 how clever a move to a specific position would be considering only the near environment. In contrast to the first network, the input units are fed with a subset of the game board

	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

(for example 5x5 fields with the centre 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 containing the situation shown in picture 2.1.2 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. 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		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

2.1.3 Third possibility

Another possibility to code a given board situation is, to divide the board in 4 sections. Then you will have to find the section where the most actions takes place. By this, I mean the section which contains the biggest variety of black, white and blank setted intersections. We calculate a value for the remaining three sections, by just representing a value for the difference between the possible stones and including a factor for free fields. We could take, for example one of the above possibilities. So we get a blurred view over the three sections. With the remaining fourth section we do not calculate at all, but set every stone into the resulting input set. One problem will gather when the board length is not eval. For example if we have a 9x9 board we have to ensure, that every section has the same size.

Also if we have similair activity in 2 sections, we will have to choose one randomly. By

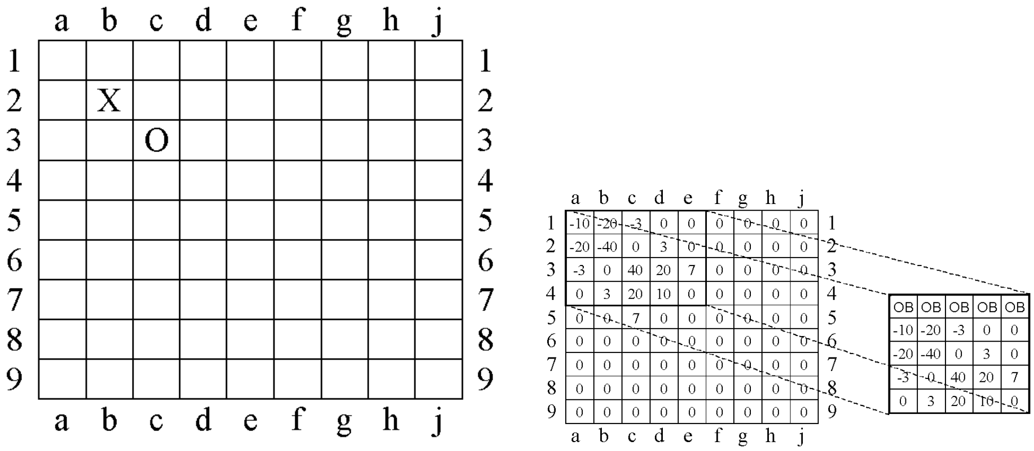

Figure 2.5: c2 is being considered for move

doing this, we want to focuss the *ANN* to play an aggressive game. We just assume, that most points can be achieved in the section were the most action takes place, instead of setting stones in the-middle-of-nowhere. In a second stage we could just set a magnifying glass on the board situation with the most action. For example, we focuss the input data on the last stone position of out opponent. We just want to keep secured the fact, that the board doesn't calculate board positions which would led to a defeat.

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 functions work.

2.2.1 First possibility

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 1000 hidden units. Both used 27 input-items and 19 output-items. Weights were initially randomized. As the learning function a standard backpropagation 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. Although

SSE was slightly better, 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 possibility

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 impoprtance 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.

2.2.3 Further enhancments for the input layer

According to 2.1.3 we tried to define a new input layer structure. After some testing we came to the result, this possibility is not an option for the input layer, but for preselection of the test field. We will filter the net through a net, and feed the output in a net which is described at 2.1.2.
We assume, that most interesting places on a game board surround the last stone set. So we will train our filter net to focus on this special part of the board.

Diese Kapitel wird noch weiter ausgearbeitet

Chapter 3

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.

Diese Kapitel wird noch weiter ausgearbeitet

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

SSE after 1000 cycles of training using momentum backpropagation technique with a training set of 576 patterns and a fully connected hidden layer.
Further investigation of the neural networks (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.
Standard Extended: similar scheme 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 and white stones. The consequence is the input layer being twice as large compared to the other two designs.

Appendix A

Report of Meetings

1st Meeting

Date: 13th March 1999

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 1999

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 1999

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 1999

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Brigitte Kendlbacher, 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 1999

Place: Institute for ComputerScience Room 2.30

Persons: Nicolas Goell, Mark Rehbichler, Brigitte Kendlbacher, 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.

List of Figures

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.