CALM, a Different Approach
Evolution in and Experimenting with CALM Architectures

Roberto Lambooy

August 1995

Master's thesis
Leiden University
The Netherlands

Abstract

CALM (Categorization And Learning Module) is a rather recent [Murr 92] neural network paradigm, developed at the Leiden University Department of Experimental Psychology, which is said to be more powerful than older paradigms, like Back Propagation Networks (BPN) and Hopfield Networks. Instead of loose nodes, a CALM-network consists of a number of modules, each consisting of a number of nodes of four different types, each with its own function. This thesis shows some research into how well CALM can do on different types of problems using different types of methods to develop architectures and parameters.

This was first done by using the method Boers and Kuiper ([Boer 92]) used in their research, followed by some manual methods and use of evolutionary strategies. This change of direction was mainly taken because the first direction did not result in any useful network architectures. Unfortunately CALM-networks do not appear to score as well on smaller problems with strict categorizations of input patterns, as might be expected from results on, for instance, digit recognition.

Contents

Chapter 1

Introduction

The main idea behind this research was to do some experiments on the relatively new artificial neural network paradigm CALM. This, of course, is a rather vague description of a research project, but this seems inherent in research. It more or less started out as a sequel to Boers and Kuiper's research ([Boer 92]). They determined the architecture of Back Propagation Networks using genetic algorithms and interpreting the thus created strings as an L-system ([Lind 68]). The L-system thus created generates the architecture of a neural network. Since this did not appear to work as well with CALM as it did with Back Propagation Networks, a somewhat different approach was chosen. The main reason for the lack of success with CALM appeared to be the fact that the parameters did not receive any attention. If these are evaluated at the same time as the architecture, the process of finding a network becomes to time assuming, at least with the current state of technological art. Therefore a couple of network architectures were chosen and a genetic method was used to determine the values of the parameters. Besides this, some side steps were taken, such as categorizing of random input and writing an interface that visualizes learning of CALM-networks.

This chapter gives a brief introduction of the main pilars for this research. The following chapters will give more in depth information about these, although they still remain introductory. For further information on these paradigms and/or methods, the cited references can be used.

1.1 Genetic Algorithms

Genetic algorithms are based on the ideas as laid out in the theory of Darwinian evolution. Darwinian evolution [Darw 1859] states that the fitter a certain member of a population is, the greater the chance the member will reproduce, and in that, will live on in his offspring. According to Darwin evolution was unintentional (as opposed to Lamarckian evolution ([Bem 89]).

In genetic algorithms a population of strings is used, where each of the members of the population can be seen as a *chromosome*, consisting of a number of *genes*. These genes are interpreted as parameters for the problem to be solved: the total of genes of one member creates one "creature" which will be used in attempting to tackle the given problem. According to how well this creature does at tackling the problem, a *fitness* is awarded. If the creature does well it receives a high fitness, if it does poorly it gets a low fitness. The fitness determines how large the chance is for the particular creature to survive and reproduce when a new generation of the population is being created. In creating a new generation, the most commonly used operators are *selection, crossover, inversion* and *mutation*. Genetic algorithms are described in more detail in chapter 2. One of the pioneers of GAs (*genetic algorithms*) was John Holland: [Holl 75].

1.2 L-systems

Living beings develop by the grace of genes. Each being has its own genetic information to start with (*genotype*), which determines the way it will look in the end (*phenotype*). This genetic information does not completely determine how the being will look, but is more a kind of *recipe* [Dawk 86]. All cells contain the same set of gene coded information, which they use during development, but their final form does not only depend on this genetic information, it also depends on its cellular environment and on the information from genes which were read in the past. All this renders the development of the cell a local matter.

As the biologist Aristid Lindenmayer wanted to model this development in plants, he developed *L-systems* [Lind 68]. By using *rewriting rules* with an L-system, a string can be rewritten into another string by rewriting all characters in the string *in parallel* into other characters. It should be noted that rewriting in parallel results in a different kind of formal language than rewriting just one occurrence at a time. L-systems will be described in more detail in chapter 3.

1.3 Neural Networks

...or maybe one should say artificial neural networks.

For a long time humans have tried to create computer programs exhibiting intelligent behaviour. For quite some time they thought that intelligently behaving computers were just a couple of years away. And then to think that we are not even certain we know what *intelligence* is...

In the early days of computers, limited attempts were made to create intelligence by deterministic programs, programs which behaviour can be predicted. A popular approach was formed by rule based systems, but these never have been able to live

up to expectations. A different approach was chosen, *reverse engineering*, which had proved to be successful in many other areas. The human brain consists of many, seemingly computationally rather simple, units, called *neurons*. After taking a closer look at these units, however, it becomes clear that neurons are far from simple. It is possible to make more or less accurate simulations of these, by abstracting from most of the more complicated mechanisms at work in a neuron, which is exactly what is done in artificial neural networks. Even so; it is at the moment impossible to imitate the brain as a whole, partly because of lack of insight in the interaction between the neurons and partly because current computers are far too slow to simulate the complexity and the number of the connections. Therefore the aforementioned large simplifications are made in artificial neural networks (ANN).

The neurons in the human brain are represented by so called *nodes* in an ANN. Each of these nodes receives input from other nodes and, based on this input, creates output, which is passed on to other nodes. Part of the nodes are *input nodes* and part of the nodes are *output nodes*. These nodes take care of the communication with the outside world. Because of their lack of communication with the outside world, the remaining nodes are often referred to as *hidden nodes*.

When we learn an ANN to solve a problem, we say it is being *trained*. Training can happen in two ways: *supervised* and *unsupervised*. In supervised training we offer the network so called *input/output pairs*. Each pair specifies an input and the desired output for that input. The network is repeatedly offered these input/output pairs, until the network has learned the problem (or until its creator has gotten fed up with it). *Back Propagation Networks* are typically trained with supervised training. CALM can be used both with supervised and unsupervised training. In this research both paradigms were used with supervised training. In unsupervised training the network is only offered the input, to which it will have to generate output by itself. The tasks performed this way are typically categorizing tasks. One of the older and more known paradigms that is often used in unsupervised learning is the Hopfield network.

Further treatises on BPN and CALM can be found in chapter 4 and 5, respectively. Finally, chapter 6 describes some methods that do not fall into any particular category, but which are of importance.

1.4 Research Goals

In their master's thesis Egbert Boers and Herman Kuiper [Boer 92] describe a method for determining the architecture of a Back Propagation Network by interpreting genetically manipulated strings as an L-system, which in itself describes the architecture of the network. Part of the work they did was developing a software package which contained, among other things, functions implementing the genetic operators and the building, running and evaluation of the Back Propagation Networks.

In this work an adaptation of their work is described that genetically determines

the architecture of a CALM network, as well as the learning parameters of the network. Since the understanding of CALM is even less—due to the extra complexity—than the understanding of BPNs, it is even more difficult to determine an optimal architecture for a CALM network than it is to determine one for a BPN.

In order to do this, several adaptations had to be made to the original functions. First of all, the Back Propagation part had to be replaced by a CALM library (written by Bart Happel). Apart from that, the genetic functions had to be adapted to not only generate strings representing the architecture of the network, but also the values of the thirty learning associated parameters CALM has.

Along the line, though, several other directions of research were taken, for instance, categorizing random input (to see what kind of problem space divisions CALM can handle), determining the parameters by adding noise (this to optimize the parameters, which clearly enhances learning), as well as visualizing the learning of CALM networks.

Chapter 2

Genetic Algorithms

One of the greater problems of using ANNs, is to determine a suitable topology. So if it poses us with problems, why not let a computer solve it? A rather brute way would of course be to simply sum up and evaluate all possible networks in an effective way. This, however, would be rather inefficient. So we need a way that takes us through the world of possible topologies in a better way. One way to do this is by using *genetic algorithms*.

Genetic algorithms (GAs) are a biological metaphor of *evolution*, as laid down by Darwin (evolution, that is) and were introduced by John Holland [Holl 75].

2.1 Overview

Goldberg [Gold 89] mentions the following differences between GAs and more traditional search algorithms:

1. GAs work with a coding of the parameter set, not the parameters themselves.
2. GAs search from a population of points, not from a single point.
3. GAs use pay-off (objective function) information, not derivatives or other auxiliary knowledge
4. GAs use probabilistic transition rules, not deterministic rules.

The parameters of the problem are usually coded in binary strings (analogous to chromosomes in biology). The coding of the parameters as well as the evaluation of chromosomes created by the GA are done outside of the GA's view. All the GA does is creating one population from another, generation after generation. Normally each of the parameters of the coded problem is represented by one *gene* in the chromosome.

During reproduction a new population of strings, possible solutions to the problem, is created by selecting and recombining strings already in the population according to their *fitness*, which is externally awarded and states how well the string does at solving the problem. This can be compared with natural selection, as described by Darwin: the fitter the organism, the greater the chance it will survive and therefore the greater the chance it will reproduce.

A fitness can be awarded in many different ways. If we for instance want the genetic algorithm to find the maximum of a function, we can simply take the function value as the fitness: the maximum of the function corresponds with the optimal attainable fitness. If the GA has to find the best neural network for a certain solution, the criterion used could, for instance, be the number of inputs associated with the desired output: the more inputs that are correctly associated with an output, the higher the fitness of the string. Again, the GA is unaware of the meaning or the origin of the fitness, it is merely a criterion to be used in selecting individuals from the population for reproduction.

The population starts out as a collection of random strings, each of which is evaluated (has its fitness calculated). As new populations are created from this, generation by generation, the overall fitness will go up, thus creating better solutions for the problem.

The four most commonly used genetic operators are *selection, crossover, inversion* and *mutation*. Each of these operations constitutes of only random bit flipping, string copying and random number generating. Crossover, mutation and inversion are all applied to a limited portion of the population. Since selection alone is not enough in the general case (some simple cases excepted), some of the other operators have to be included.

2.2 Selection

Selection is used to choose strings from the population for reproduction. The higher the member's fitness, the greater the chance of being picked for reproduction. There are two important selection methods, of which only the latter was used in this research.

The first method is *roulette wheel selection*: strings are selected with a probability proportional to their fitness. A drawback of this method is that an early highly fit member can get to dominate the population, due to its high selection chance. The second is *rank based selection* where the chance of being selected is determined as a linear function of the member's position in the population sorted by fitness [Whit 89]. For this method to work the population has to be sorted by fitness. One advantage of this method is that there is less dominance by an early good solution that is not optimal and which could lead the system into a non-global optimum. However, one does have to be cautious on choosing a fitness criterion: in roulette wheel selection it doesn't really matter exactly what the order of the members is, since more or less

the same fitness leads to more or less the same chance, with rank based selection, however, the order is important. This can become a problem when all the members have fitnesses that are close together. This typically happens when fitnesses are chosen to have a too high nominal value, resulting in the differences between individuals being dwarfed by the value. This can be solved by chosing a fitness function that makes optimal use of the [0...MaxFitness] interval, with MaxFitness being the maximum possible fitness.

2.3 Crossover

The crossover operator creates a new member for the new population by combining different parts from two members of the previous population. A number of *crossover points* is chosen at random. The new string is made from alternating parts of the two originating strings: the bits from the first string are copied till the first crossover point is reached, than we switch to the second string, at the first bit following the crossover point. This string is copied till we reach the next crossover point, where we switch back to the first. If more crossover points are used –which was not done in this research– the crossing over between strings goes similarly. An example is shown in figure 2.1(a).

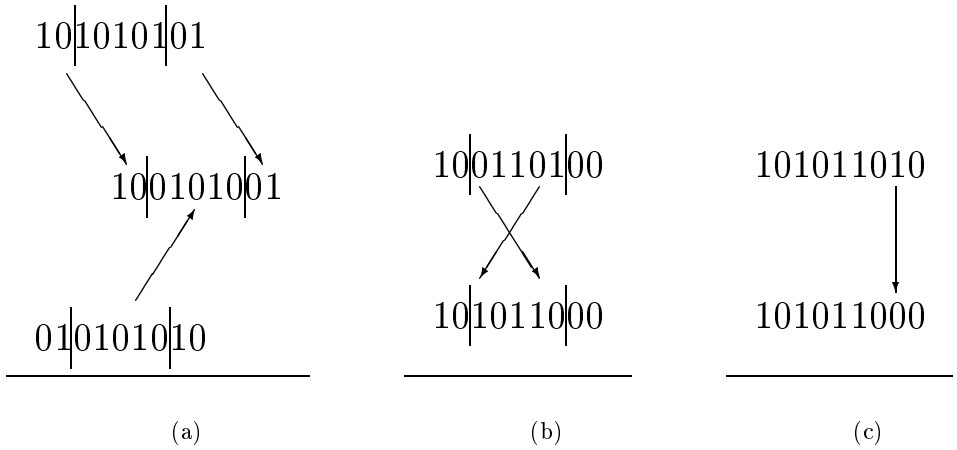


Figure 2.1: Examples of Crossover, Inversion and Mutation

2.4 Inversion

Inversion is an operator that swaps around parts of the originating strings. Two inversion points are chosen at random. The genes between these two points are swapped, the first becoming the last, the second the penultimate, the penultimate becoming the second and the last becoming the first, etcetera. An example is shown in figure 2.1(b).

As mentioned before, a member consists of several genes, each coding part of the recipe for the eventual organism. It would be both unwanted and biologically strange to simply intermingle two genes coding distinct information. Therefore steps have to be taken to prevent this from happening.

2.5 Mutation

Mutation is as simple as you can get in genetic operators. All it does is flip one or more bits in the string from 0 to 1 or from 1 to 0.

The purpose of mutation is to introduce new solutions in the population that are close to solutions that are known to do well already (since just one bit is flipped, the string stays largely the same). Mutation can also reintroduce genes that were lost by unfortunate cross-overs. To prevent mutation from messing the population up (by flipping too many bits, thus making the population more or less random noise), the mutation rate has to be kept low. There are applications where selection and mutation as operators are enough to solve a problem (see for instance [Gari 90]). Figure 2.1(c) shows an example.

2.6 Implementation

In his program GENITOR Whitley [Whit 89] uses a *one-at-a-time selection and replacement* schedule: a new member of the population replaces the member with the lowest fitness, thus making sure of a monotonically increasing total fitness. This results in the member with the highest fitness always staying in the population. This model is known as the *static population model*. One-at-a-time replacement is always better than creating a whole new population when it can be guaranteed that a global maximum can be reached from a good string in the population without passing a local maximum on the way.

At the Department of Experimental and Theoretical Psychology at the Leiden University, a C-library was written to create and manipulate binary strings either using roulette wheel selection or rank-based selection and replacement ([Happ 92] and [Murr 92]). The functions in this library were first adapted by Boers and Kuiper for their research, and later by me for my research. Boers and Kuiper's functions were meant to genetically determine the architecture (topology) of the network. Since not

only the architecture, but also the learning parameters were determined genetically for this research, the genetic operators, as well as the members, had to be adapted to include these.

2.7 Evolutionary Strategies

Genetic Algorithms are not the only genetic metaphor used in computation, there are others. One is Evolutionary Strategies. The variation used in this research works with a kind of random noise (as can be found in the undirected evolution as described by Darwin). In this case it was used to generate values for the parameters of CALM-networks. All parameters are multiplied by a random real value in the range $[0.50 \ldots 1.50]$, thus repeatedly generating new members. The size of the population is arbitrary, but the search tends to get to directed for small populations (in the same way as with GAs).

Chapter 3

L-systems

3.1 Biological Development

The development of living organisms is governed by genes. Each cell contains genetic information (*genotype*) which determines the way the organism will look (*phenotype*). The genetic information is not a blueprint, but more a recipe of the final form. This recipe is not followed by the whole organism, but by each cell separately. The shape and behaviour of a cell depend on the genes from which information is extracted, which in itself depends on from which genes information was read in the past as well as the environment of the gene. This leaves the development a solely local matter.

3.2 Simple L-systems

L-systems were introduced in 1968 by Aristid Lindenmayer [Lind 68] in an attempt to model the biological growth of plants. An L-system is a parallel string rewriting mechanism, a kind of *grammar*.

A grammar consists of a starting string and a set of *production rules*. A production rule consists of two parts; the part before the $\rightarrow$, the *left side*, and the part after the $\rightarrow$, the *right side*. The following production rule rewrites an A into AB. Repeated application will result in the string AB^n, with n the number of times the rule was applied.

$$A \rightarrow AB$$

The starting string, also known as the *axiom*, is rewritten by *applying* the production rules: each production rule describes how a certain character or string of characters should be rewritten into other characters. For a production rule to be applied, it must first *match*: the left side of the rule has to be the same as a part that is to be rewritten. Whereas in other grammars production rules are applied one-by-one

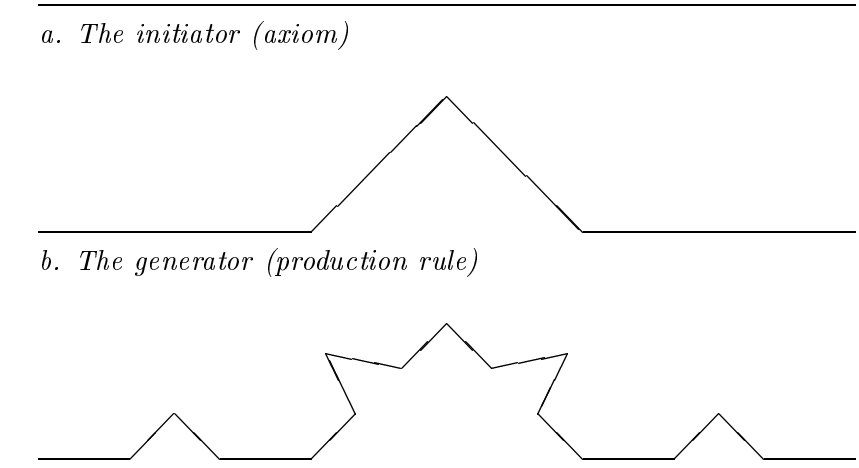


Figure 3.1: Example of a Koch fractal

sequentially, in an L-system all characters in a string are rewritten in parallel to form a new string.

This creates the opportunity to draw approximations of certain types of fractals, by using a special interpretation of a string generated by an L-system. We do this by using a LOGO-style turtle [Szil 79].

Consider the following L-system:

Axiom	F
Production rule	F → F - F ++ F - F

If we interpret F as a step forward, - as a 45 degree turn left and + as a 45 degree turn right, we could visualize this production rule as shown in figure 3.1.

3.3 Bracketed L-systems

A disadvantage of the turtle symbols from the previous paragraph, is that they can only make so called *single line drawings*, in contrast to what is observed in natural life branching, as seen in for instance plants. To give the turtle the freedom of movement to create this more natural branching, two new symbols are added:

[Remember the current position and direction of the turtle (push).

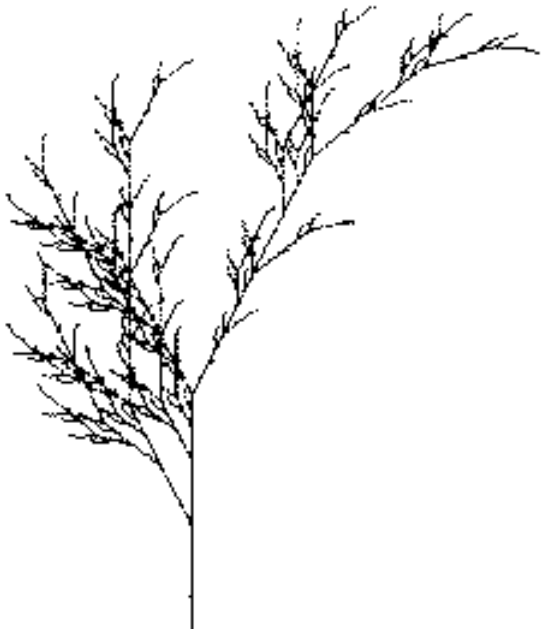


Figure 3.2: 30 rewriting steps, $\delta = 16°$, axiom F1F1F1. See the text for the production rules used.

] Restore the last stored position and direction (pop).

With these two new symbols, far more realistic drawings can be made, as can be seen in figure 3.2.

3.4 Context Sensitive L-systems

Another way of making more complex, more natural looking drawings of plants, is implemented by using *context*. Context models the exchange of information between neighbouring cells, it can be left, right or both for a certain string. An L-system without context is called a 0L-system, an L-system with one-sided context a 1L-system, and one with context on both sides a 2L-system. A production rule is of the following form:

$$L < P > R \rightarrow S$$

Where P (the *predecessor*) models the left side in the earlier production rule without context, and S (the *successor*) the right side. L and R are the *left-context* and *right-context* respectively.

If in a rule P has context on both sides, it can only be replaced by S if it has its left context directly on the left in the string and its right context directly on the right. If two production rules qualify for application, the one with context is chosen.

If we take the following production rules:

$A \rightarrow X$
$B \rightarrow Y$
$C \rightarrow Z$

the string ABC would be rewritten to XYZ, after which neither of the rules applies.

If we were to take these production rules:

$$\begin{array}{cccccc} & & A & \rightarrow & X \\ & & B & \rightarrow & Y \\ Y & < & C & \rightarrow & Z \end{array}$$

the string ABC would be rewritten to XYC in one step. The C is not rewritten because the left context is not Y at the moment of writing (remember, rewriting goes in parallel, so C's left context still is B). However, if we were to rewrite XYC we would find one rule that applies: since C's left context now has been changed to Y, the third rule does apply. This results in XYC being rewritten to XYZ.

Determining what the context is, is a little more tricky with bracketed 2L-systems. Since the left and right context is not always direct left or right from the string or character that is to be replaced, but can be distanced by a bracketed pattern (these bracketed patterns would represent branches if we were to plot the string as a tree) ([Prus 89]). If we, for instance, had a production rule with the following left side:

$$BC < S > G[H]M$$

It could be applied on the S in:

$$ABC[DE][SG[HI[JK]L]MNO]$$

skipping DE on the left side and $I[JK]L$ on the right side in the process, since these represent (parts of) branches that are of no importance to the rule to be applied.

The following example was generated by Hogeweg and Hesper [Hoge 74] and produces a quite natural looking drawing of a plant. See figure 3.2 for a picture.

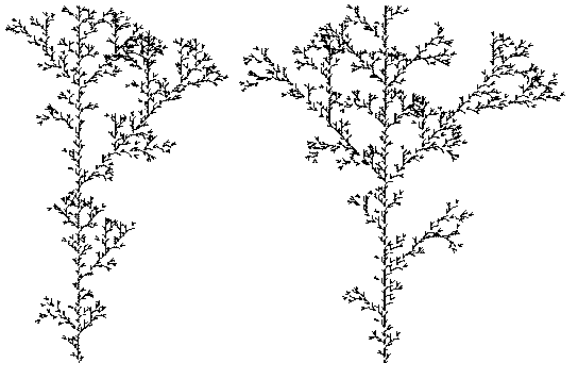


Figure 3.3: Both plants were created with the rules $F \rightarrow F[+F]f[-F]F$, $F \rightarrow F[+F]F$ and $F \rightarrow F[-F]F$, each with a 33% probability.

0	<	0	>	0	→	0
0	<	0	>	1	→	1[-F1F1]
0	<	1	>	0	→	1
0	<	1	>	1	→	1
1	<	0	>	0	→	0
1	<	0	>	1	→	1F1F
1	<	1	>	0	→	1
1	<	1	>	1	→	0
		+			→	–
		–			→	+

3.5 Implementation

Przemyslaw Prusinkiewicz and James Hanan [Prus 89] present a small L-system program for the Macintosh (in C). To experiment with L-systems Boers and Kuiper ported this to PCs. Besides fixing some"irregularitie" the program was rewritten in order to accept less rigid input files. Two features were added: *probabilistic production rules* and *production rule ranges* (both from [Prus 89]).

With probabilistic rules more than one production rule for the same L, P and R can be given, each with a certain probability. When rewriting a string, one of the rules is selected proportional to its probability. This results in more natural looking plants, without them losing their characteristic appearance. Figure 3.3 shows a plant that was created with one set of probabilistic rules.

Production rule ranges introduce a temporal aspect to the L-system and tell which rules should be looked at during a certain rewriting step. This can be used for exam-

ple, to generate twigs first and then the leaves and flowers at the end of those twigs. All the examples shown were generated using software by Egbert Boers. See section 7.1 for an extension of L-systems producing Artificial Neural Network architectures.

Chapter 4

Neural Networks

4.1 The Human Brain

The human brain consists of a large number of interconnected *neurons*. Each of these neurons shows rather complex bio-electrical and bio-chemical behaviour, although it does not appear to do more than determine whether there is enough stimulation for it to fire. What happens is that the neuron gets electrically charged substances, which are collected in the cell body. If the potential of the substances in the cell exceeds a certain *threshold* the neuron fires and the cell is neutralized again. When the neuron is not stimulated enough to fire, the electrically charged substances also float out of the cell body. Even though all the units perform such in principle simple actions, the total creates behaviour more complicated than we can fully understand at the moment (or ever...). Since the units do not appear to be the ones performing complex behaviour, it can only be the interaction between them that does. Even though neurons are rather slow compared to modern day computers, their large number gives them such a great advantage, that computers haven't managed in coming close to the human brains in their achievements. It is this larger number of the neurons that is believed to be the basis of intelligence, even though we do not even know what exactly intelligence is. And if we ever find out what intelligence is, it will probably be done by intelligence...

4.2 Artificial Intelligence

Those working in the field of Artificial Intelligence are mainly concerned with trying to create intelligent behaviour in machines. This task is a task into the unknown for more than one reason, the first being that we are not even that sure we know what intelligence is. Webster's Dictionary alone gives four definitions of Artificial Intelligence:

1. An area of study in the field of computer science. Artificial Intelligence is concerned with the development of computers able to engage in human-like thought processes, such as learning, reasoning, and self-correction.

2. The concept that machines can be improved to assume some capabilities normally thought be like human intelligence such as learning, adapting, self-correction, etc.

3. The extension of human intelligence through the use of computers, as in times past physical power was extended through the use of mechanical tools.

4. In a restricted sense, the study of techniques to use computers more effectively by improved programming techniques.

Alan Turing [Turi 63] has proposed a test that should be satisfied in order to speak of intelligence, the *Turing test*. A person is placed in a room with two terminals, each of which he can ask questions. On one of the terminals the answers are given by a human, on the other one by a computer. If the person fails to determine which of the two is manned by the computer, the computer is said to have passed the test. Of course the set up has to be made in a way that makes it impossible to determine which is which by factors not of importance, such as the speed of answer (a computer is unlikely to make typing errors, humans however...).

Traditionally researchers tried to create intelligent behaviour by using things like rule based systems, but these never could give the desired behaviour. So far no one came up with a set of rules that created intelligent behaviour, although there are some rule based systems that perform pretty well in a very small field—so called *expert systems*—but don't ask one of these how much a carton of milk costs... It should be noted that ANNs can also be seen as rule-based systems, however, as opposed to "traditional" rule-based systems, ANNs work sub-symbolically.

Another problem in these rule based systems is that the programmer has to supply all the knowledge. However, it turns out that the real life expert often doesn't know exactly why he came up with a certain solution for a presented problem. Part of being an expert is "feeling" the solution. This method of problem solving is, of course, hard to formalize. Therefore a new area of research, *knowledge engineering*, has arisen which purpose it is to find methods for acquiring knowledge.

The lack of success in the search for intelligence with traditional methods, has made researchers look for other options. They found one in *reverse engineering*, a method that worked well on a lot of other occasions. The idea is to look at something that works and try to imitate that. In this case that means trying to make something that works like a human brain.

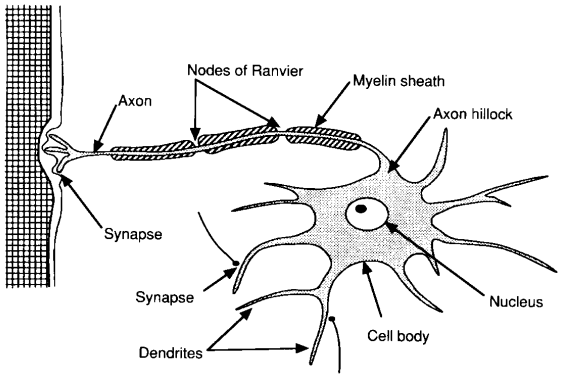


Figure 4.1: A Neuron

4.3 The Neuron

The main actors in our brains are neurons. Therefore artificial neural networks are based mainly on the workings of neurons. The neuron shown in figure 4.1 can be separated in the three functional parts: *axon, cell body* and *dendrites*. The dendrites receive information from other neurons and transmit those signals by electro-chemical means to the cell body. The body collects all these signals, sums them and if this sum exceeds the threshold for that neuron, the axon is activated: the neuron *fires*. The axon transmits this activation to the dendrites of other cells. The communication from one neuron to another takes place at the *synaptic junction*, or *synapse*. The synapse is a small gap between the axon of one neuron and a dendrite of the next. The axon releases *neurotransmitters* which are absorbed by the dendrite. These chemical substances influence the potential inside the receiving cell body. When learning takes place, the amount of released neurotransmitters changes (for a more elaborate coverage of these concepts see [Free 91] and [Bloo 88]).

4.4 The Connected Neurons: a Brain

The brain consists of about 10^{11} neurons. Because of this number it is virtually impossible to simply connect every neuron to every other neuron. To prevent from having to fully connect, the brain got split in several regions, most of which are split in several regions again. The smallest neuron structures found in the brains are the so called *mini-columns* ([Bloo 88]), consisting of some 100 neurons each. The simplest division of the brain is in the left and the right hemisphere. These two hemispheres

are connected by the *corpus callosum*. Apart from this, there is also a division into smaller *functional areas*, such as the visual area or the auditory area. Usually there are relatively few connections between areas with different functions. This strong modularity is partly suggested by patient studies, see for instance [Gazz 89]. Despite this strong modularization the brain still has some 10^{15} connections. This number immediately indicates one of the reasons researchers have as yet not run complete simulations of human brains—apart from the fact that they are not that sure that our current ANNs are a fair copy of our brains—it would be too big a simulation to fit into a computer's memory (apart from the fact that it takes too much time).

From these numbers we can give an estimate of the brains computing power. Every axon is able to transmit one pulse every 10 milliseconds, and since a neuron fires or not, we can say that one neuron transmits 100 bits per second, which results in the complete brain transmitting 10^{13} bits per second. Jacob Schwartz [Schw 88] estimates the total number of arithmetic operations even higher, at 10^{18} per second, needing 10^{16} bytes of memory. This is a lot faster than the best of current supercomputers does; the best we can do so far is the CM-5. This machine is developed to entail 16,384 processors, giving $0.7 * 10^{12}$ floating point operations, with a transport of $1.6 * 10^{11}$ ([Hill 93]), leaving us still about a factor 10^6 short. It should be noted that the largest version sold so far had "only" 1,024 processors. This gap between current and needed possibilities is a definite motivation to do a lot of research in *massive parallel computing*

4.5 Artificial Neurons

As a consequence of the huge complexity of the human brain and the state of current hardware, it is impossible to build an artificial brain that imitates the natural brain in all its detail. So in order to make use of its functional principles, we are forced to make very large simplifications regarding the computations performed by the neurons and their connectivity.

The most obvious function of neurons is taking their real-valued inputs and determining their activation from that. This activation determines whether the neuron will fire or not. Normally the stimulation is simply the (weighted) sum of all the inputs. Usually a bias is added, which shifts the activation relative to the origin:

$$stim = \sum_{i=1}^{n} w_i x_i + \theta.$$

The analogy with the real life neuron is obvious: where the neurons in the brain get charged chemicals in their cell bodies, where reaching a charge above a certain threshold makes the neuron fire, the artificial neuron simply sums the input values and determines whether it will fire by comparing it to a threshold value. The amount of

neurotransmitters transmitted by the axon are a metaphor for the weights in artificial neurons. The connections in artificial neural networks are not specifically excitatory or inhibitory, like they are in the brain, but the weights can be either negative or positive, and can be changed during the learning process. It should be noted that the stimulation is not necessarily the same as the activation of the neuron:

$$act = f(stim).$$

Sometimes it is implemented as a function of the stimulation and the previous activation:

$$act(t) = f(act(t-1), stim(t)).$$

Although f is determined for a large part by the type of ANN being used, the basic functioning of neurons is globally the same, since all ANNs are in one way or another based on the original brain.

4.6 Artificial Neural Networks

In artificial neural networks we take the next step: the connection of a number of neurons into a network. One of the main problems to be tackled in artificial neural networks is that we are modeling things we do not fully understand. So we don't really know whether the neural network we create is even remotely similar to its original. What we do know is that one of the larger advantages of ANNs is that we do not have to present to a network how we came to a certain solution: we simply present the network with a lot of problems and their solutions, and the ANN finds the regularities in the associations of input/output pairs itself.

Some areas where neural networks have been successfully used are ([Hech 90] gives a more extensive overview):

- handwritten character recognition,
- image compression,
- noise filtering,
- broomstick balancing,
- automobile autopilot,
- nuclear power-plant control,
- loan application scoring,
- speech processing,

- medical diagnoses .

4.7 The Training Set

Since it is normally impossible to present a network with all possible inputs, we only present it with part of it, the *training set*. This set has to be chosen in such a way that the network also gives correct output for an input that was not in the training set. If the network also responds well to inputs that were not in the training set, it is said to *generalize* well. If the training set wasn't a good representation of all possible inputs, the network probably will not perform too well on inputs not in the training set. Generalization is quite similar to *interpolation* in mathematics.

4.8 Back Propagation Networks

Probably the best known artificial neural network learning paradigm is Back Propagation. It was first formalized by Werbos [Werb 74] and later by Parker [Park 85] and Rumelhart and McClelland [Rume 86]. It is a *multi-layer feed forward* network that is trained with *supervised learning*. Normally a BPN has an input and an output layer, and a certain amount of hidden layers. The input and output layers are mandatory, the number of hidden layers is free, but often a single layer is chosen. Nodes in a certain layer only get input from other lower layers, which means that the input layer does not get any input from within the net, and only give output to nodes in higher layers, which means that nodes in the output layer do not give output to other nodes. This constitutes the feed forward principle: input only comes from lower layers and output only goes to higher layers. Their is no recurrence in a feed forward network, although there are some adaptations of the BPN paradigm that allow a limited amount of recurrence. It should be noted that the lack of recurrence is a huge simplification of the real brain. Figure 4.2 shows an example of a BPN with one hidden layer. The subsequent layers are fully connected.

During supervised learning the network is repeatedly presented input/output pairs (I, O) by a supervisor, where O is the desired output of input I. The input/output pairs specify the activation patterns of the input and output layers of the network respectively. The network has to find an internal representation that associates the input with the desired output. To achieve this, Back Propagation uses a two-phase *propagate-adapt* cycle.

In the first phase the network is presented with the input and the activation of each of the nodes is *propagated* through the net to the first hidden layer (or the output layer, if no hidden layers are present), where each node sums its input and decides whether it should fire to the modules in the next layer. This process repeats itself until the activations have reached the output layer.

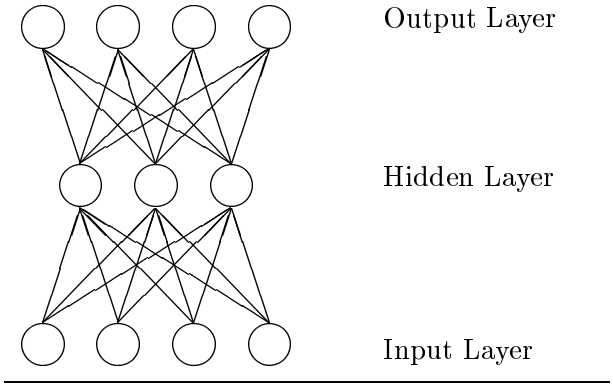


Figure 4.2: A Back Propagation Network

In the second phase the output of the network is compared to the desired output and the error is calculated for each of the nodes in the output layer with this formula:

$$\delta_i^o = o_i(1 - o_i)(y_i - o_i)$$

where o_i is the output the network gave on node i, and y_i is the desired output for node i. These error values are transmitted to the last of the hidden layers—hence the name Back Propagation—where for each node its total contribution to the error is calculated:

$$\delta_i^h = h_i(1 - h_i) \sum_{j=1}^{r} \delta_j^o w_{ij}^o,$$

where w_{ij}^o is the weight of the connection from hidden node j to output node i. The calculations for possible other hidden layers are done in a similar way. Based on these contributions to the errors, the connection weights are *adapted*:

$$w_{ij}^o(t+1) = w_{ij}^o(t) + \alpha\delta_j^o h_i + \beta\Delta w_{ij}^o(t-1)$$

with $\Delta w_{ij}^o(t-1) = w_{ij}^o(t) - w_{ij}^o(t-1)$, for the weights to the output layer and

$$w_{ij}^h(t+1) = w_{ij}^h + \alpha\delta_j^h x_i + \beta\Delta w_{ij}^h(t-1)$$

for the hidden layer(s). This makes the overall error:

$$E = \frac{1}{2}\sum_{i=1}^{r}(y_i - o_i)^2$$

(for this input/output pair) smaller, with the overall objective being to reach its minimum.

When we take a $(n+1)$-space, with n the number of weights in the network, we can plot the total error of the net for all inputs as a function of all the weights. This space is called the *error space*. Since we want the network to perform as well as possible, we want to find the minimum in this error space. The function drawn in this space can be seen as a surface across which we let a marble roll during learning with Back Propagation: it always follows the steepest gradient, or the direction that goes down as fast as possible. However, it is possible that the error surface not only has the wanted global minimum, but also some local minima. The fact that a certain point on the error surface is a minimum, means that the surface goes up on all surrounding sides. This means that when the marble hits a minimum, it will stay there. This is—of course—rather unfortunate when the found minimum is not a global, but a local minimum: the network gets stuck in the local minimum.

4.9 Problems with Back Propagation

As mentioned in the previous paragraph, one of the problems of Back Propagation is that it can get stuck in a *local minimum*. This isn't too bad if the local minimum turns out to be close to the global minimum, but there is no guarantee of that. This problem can be partially solved by using a *momentum* term. The momentum term uses the speed the marble already has, so when it hits the minimum it will not immediately remain there, it will first go up again. This works because the edges around a local minimum are lower than those around a global minimum most of the time (the global minimum is the deepest point of the problems space, so speaking on the average, it must be harder to get out of a global minimum, than it us to get out of a local minimum). So if the momentum term is chosen right, it can push the marble out of the ditch created by the local minimum, but it will remain in the ditch created by the global minimum. This momentum term also enhances learning, since it uses the steepness of the slope the "marble" is on, instead of simply using steps of fixed size when moving through the error space.

Another problem associated with Back Propagation is that the place at which one starts on the error surface — which is determined by the initial weight settings, which are often random — determines whether or not a good or the best solution is found. When a solution is found that performs well on the training set, the network might still perform badly on the overall set of input, if the training set was not a good representation. One danger in Back Propagation is for the network to get *overtrained*.

This means that the net did not look at similarities over the input, but simply learned all associations by heart. If presented with output not in the training set, the network will likely respond with other output than the desired. This problem only occurs when the network is still further trained, even though it already gives correct output. The network has learned to detect global features at first, but is trained longer and reaches such a specialization in the given training set, that it loses its ability to generalize: there is no need for generalization, it already knows every input/output pairing. This will have resulted in perfect scores on the training set. Overtraining can only happen if the network is large relative to the training set. In this case training is better stopped before full conversion on the training set is reached. Other ways to prevent overtraining is using a smaller network or adding noise to the input. Both methods will result in poorer performance on the training set, but will lead to better overall results.

Unfortunately Back Propagation does not do well on *extrapolation*. If it is trained in a certain area, it does not perform well in other areas, even if these are close to the trained area. This stresses the importance of choosing a proper training set. Back Propagation can be used, however, to make predictions when historic data is available.

A last problem is the occurrence of *interference*. This occurs when a network is supposed to learn similar tasks at the same time. Apart from the fact that smaller networks are unable to learn too many associations—they simply are full after a certain amount of learned associations—there is also the danger of input patterns being so hard to separate, that the network can't find a way to do it. When we look at the problem we can take its input and divide that into categories. If we plot this, we would get a problem space. The network has to fill this space with figures of such a form that all inputs from the same category are included in the same figure. This means that the network has to encode these forms in some way. The more complex these forms—or the more precise they have to be—the harder it is for the network to learn it. This means that inputs that are close together, with little room left for a line separating the categories of the problems, and figures with strange forms (the more concave, the worse) are hard to learn. An example of such interference between more classifications is the recognition of both position and shape of an input pattern [Ruec 89]. Rueckle et al. conducted a number of simulations in which they trained a three layer Back Propagation Network with 25 input nodes, 18 hidden nodes and 18 output nodes to simultaneously process form and place of the input pattern. They used nine, 3x3 binary input patterns at 9 different positions on a 5x5 input grid, resulting in 81 different combinations of shape and position. The network had to encode both form and place of a presented stimulus in the output layer. It appeared that the network learned faster and made less mistakes when the tasks were processed in separated parts of the network, while the total amount of nodes stayed the same. The number of hidden nodes allocated to both sub-networks was

of importance. When both networks had 9 hidden nodes the combined performance was even worse than that of the single network with 18 hidden nodes. Optimal performance was obtained when 4 hidden nodes were dedicated to the place of the pattern and 14 to the apparently more complex task of the shape of the pattern. It should be emphasized that Rueckle et al. tried to explain why form and place are processed separately in the brain. The actual experiment they did, showed that processing the two tasks in one unsplit hidden layer caused interference. What they failed to describe, however, is that removing the hidden layer altogether, connecting input and output directly, leads to an even better network than the optimum they found using 20 hidden nodes in separate sub-networks, as described in [Boer 92]. However, more recent research has been unable to reproduce this (see Lawrence Pit's Master's thesis, which will appear around September 1995). Another example of the difficulty Back Propagation has in separating a task in modules can be found in Norris' article about an idiot savant date calculator [Norr 90].

The problems mentioned, however, do not occur solely with Back Propagation, a lot of other network paradigms suffer from it. This brought on the search for *modularity*, which we already find in the brain, and which we also find in CALM, to which I will turn in the next chapter.

4.10 Modular Back Propagation

Until now we have discussed only simple networks, where every layer is fully connected to the next. However, this is not due to a limitation in the Back Propagation's learning rules. More complicated networks are created by, for instance, adding hidden layers. This doesn't really add to the computational power of the network—in fact, it has been proven that all continuous functions that can be implemented by a network with more than one hidden layer can also be implemented by a network with one hidden layer, although we would need an infinite number of nodes in the hidden layer for the error to approach 0—but it does enhance the speed with which the network learns, especially for highly nonlinear inputs, inputs that are hard to differentiate.

When more hidden layers are used, all layers are still fully connected to the next. This means that all modularity in the net has to be propagated to the nodes through the connections. Another way is to not simply create full connectivity between layers, but leave specific connections out. So by adding hidden layers without full connectivity, we can greatly enhance the amount of modularity in the network, without raising the number of weights to astronomical numbers. See for instance the accompanying figure 4.3(a) , which can be separated in two parts.

Since there are no connections between the two parts of the network, the number of weights is reduced by 10 compared to a fully connected network with the same number of nodes in each of the layers. Apart from a speed up caused by less connections, there might also be a speed up due to the greater modularity of the network.

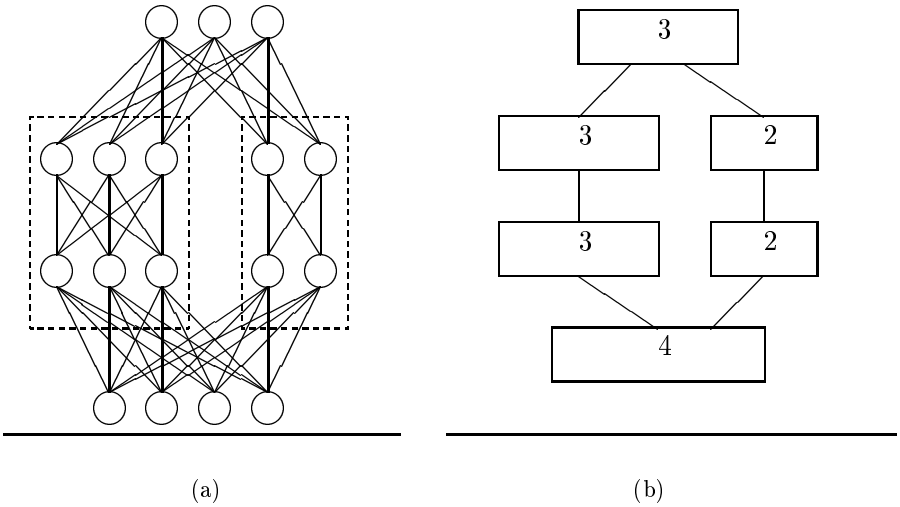


Figure 4.3: A modular Back Propagation Network and its modularized counterpart.

To further enhance this idea of modularity, we define a *module* to be a group of mutually unconnected nodes with as well the same set of input as of output nodes [Boer 92]. This means that a fully connected network has the same number of modules as it has layers. The network from figure 4.3(a) has 6 modules. Every node in a Back Propagation Network is in exactly one of these modules. In figure 4.3(b) the network of figure 4.3(a) is remodeled to this standard, and is, as you will see, a lot simpler than its original counterpart.

This idea of modules is much more elaborated on in CALM, to which we will turn in the next chapter. It should be noted that there are connections with a CALM, but that they are not between the CALM-counterparts of BPNs nodes, but between nodes of different types.

To test whether this modularization worked, Boers and Kuiper [Boer 92] implemented a Back Propagation Network for the XOR-problem with a different topology than the network used by Rumelhart and McClelland [Rume 86]. The two networks are shown in figure 4.4. Rumelhart and McClelland's network got stuck in a local minimum a couple of times during their experiments. Boers and Kuiper found that their network not only always learned to solve the problem, but it also learned faster than Rumelhart and McClelland's network did. On plotting the number of training steps required by the net as a function of the two learning parameters Boers and Kuiper found a much more regular dependence for their network than they found

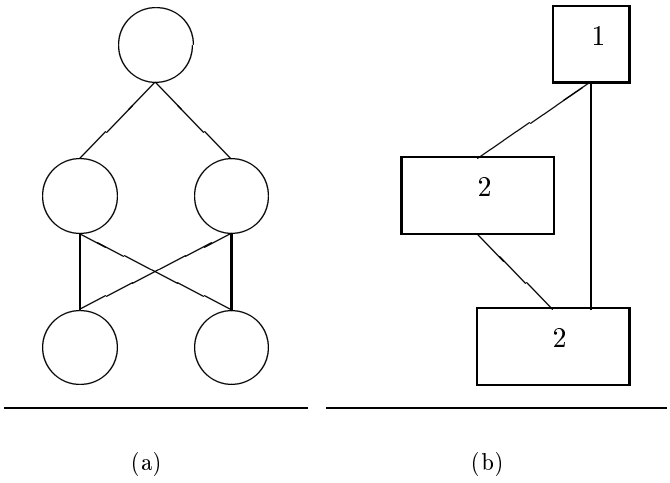


Figure 4.4: "The McClelland and Rumelhart" and "The Boers and Kuiper" networks

for Rumelhart and McClelland's network. Apart from that, they found that for the best combination for the learning parameters the Rumelhart and McClelland network needed 1650 training cycles, while the Boers and Kuiper network needed only 30 cycles. McClelland and Rumelhart also found a net with a different architecture, which did work with all experiments, although it still tended to be slow in learning.

The intuitive idea behind imposing modularity on a network is modeling the weight space in such a way that all local minima disappear, making the surface a lot smoother along the way. This greatly enhances learning. But if modularity is imposed by simply changing the structure of the network, we get our network in a highly unnatural way. What we're trying to do, is to imitate nature by creating an artificial neural network, but since neural networks aren't structured by some divine hand, but develop naturally, this is not *biologically plausible*: it is not similar to the way things happen in a more natural context. So we need a different method if we want to preserve biological plausibility.

Another problem modularity poses us with, is that of finding a suitable topology for a given problem (in determining whether a network is good, we also need to know what problem it is supposed to solve, since different problems have different suitable topologies). This was already a problem with classic networks, but with the extra complexity of the network connections, this problem gets even more difficult to solve, especially by hand.

The simplest method is of course to choose something for ourselves, however, this

is—as mentioned before—hardly biologically plausible and is difficult to do, since we often do not know what would be a good topology for a network.

Since BPNs were of secondary importance to this research, only a limited overview was given here. More complete treatises on them can be found in [Boer 92] and for instance [Free 91] and [Hech 90].

Chapter 5

CALM

A relatively recent approach to artificial neural networks, CALM, was developed at the Department of Experimental Psychology at the University of Leiden [Murr 92]. CALM stands for *Categorizing And Learning Module*. A CALM is a building block for a network of CALMs, but is far more complicated than the building blocks in for instance Back Propagation or Hopfield networks. A CALM consists of several types of nodes (these can be compared to the nodes in other types of neural networks) connected to each other by connections with fixed weights, that is the weights stay constant during learning.

The idea of a CALM is, as its name already suggest, to categorize input. Therefore the chosen convergence criterion is often for exactly one node to remain active, the node that represents the category in which the input is put.

5.1 The Maths behind CALM

The effective input to node i—the excitation e_i—is the weighted sum of the individual activations of all nodes connected to the input side of node i. Each input is either excitatory or inhibitory (this is a feature of a connection, not of a node). It should be noted that this is highly biologically plausible, since the neurons in the human brain also only have connections which are either positive or negative. The activation of node i at time $(t+1)$, $a_i(t+1)$, is a function of $a_i(t)$, the activation at time t, and its input excitation e_i, expressed in the following formula:

$$a_i(t+1) = (1-k)a_i(t) + \frac{e_i}{1+e_i}[1-(1-k)a_i(t)], e_i \geq 0$$

for excitatory input, and:

$$a_i(t+1) = (1-k)a_i(t) + \frac{e_i}{1-e_i}(1-k)a_i(t), e_i < 0$$

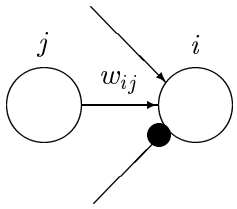


Figure 5.1: The labeling of nodes

for inhibitory input, where

$$e_i = \sum_j w_{ij} a_j(t)$$

with w_{ij} denoting the weight of a connection from node j to node i (see figure 5.1). The difference in time between t and $t+1$ is called the iteration time. k, the decay factor, is a constant between zero and one. It should be noted that the summation for e_i takes both inhibitory and excitatory input into account.

The activation rule can be split into three important parts. The first, $(1-k)a_i(t)$, represents the decay of the activation of the node. When no new input arrives, the activation of the node gradually decays to zero (asymptotically). Since k is supposed to be a decay parameter, it should be chosen between 0 and 1; if it is smaller than 0 the activation will rise with time, if it is chosen larger than one, the activation will flip signs every iteration (which is something we do not want to happen). The second part, $\frac{e_i}{1+e_i}$, for $e_i \geq 0$, squashes the input excitation to a number between one and zero. The third part $[(1-k)a_i(t)]$, makes sure the increase of the excitation decreases with growing excitation, thus making the activation grow asymptotically towards the maximum activation. Something equivalent holds for the part for $e_i < 0$: $\frac{e_i}{1-e_i}$ squashes the negative excitation between minus one and zero, the $(1-k)a_i(t)$ component ensures the asymptotic approach of the minimum activation.

5.2 The Nodes

There are four types of nodes in a CALM. This categorization is made on both the type of connections of the nodes to other nodes, and the function each type of node

has within—or, as we shall see, outside—a CALM.

The first type is comprised of nodes which have modifiable connections to similar nodes in other modules. All the outgoing connections from *R-nodes*, short for Representation nodes—both inside and outside the CALM—are excitatory. Since the activation of these nodes represents the given input or the resulting output of the CALM, these nodes are called *R-nodes*. R-nodes are the only nodes that communicate with the nodes outside of the CALM.

The nodes in the second category are the *V-nodes*, or Veto-nodes. V-nodes have only inhibitory outgoing connections. A V-node inhibits all the other nodes within the CALM: they inhibit all their fellow V-nodes, as well as all R-nodes. Every V-node receives excitatory input from exactly one R-node, and every R-node receives inhibitory input from all V-nodes, but a little less from the V-node to which it is paired. This means that each R-node forms a pair with a V-node. Since such a pairing is mandatory, there is always the same number of R-nodes and V-nodes.

There is always exactly one node of the third category in a CALM, the *A-node*, or Arousal-node. The A-node is excited by all R-nodes and inhibited by all V-nodes. Because of the weights on the connections within the CALM the excitation of the A-node designates the amount of competition within the module. In a CALM competition will be most prevalent for inputs that weren't offered before, new input which hasn't been learned by the network or CALM yet.

The fourth and final node category is the *E-node*, or External-node. It is a node that differs from the other types of nodes in that it isn't exactly in the CALM, even though it does belong with the CALM. It is meant to model a certain part of an arousal center outside the model. This is a metaphor to certain sub-cortical centers in the nervous system that seem to diffusely activate large parts of the cortex. Several E-nodes may be connected to form such a center, but in most simulations just one E-node per CALM is used. So generally an E-node has connections to just one CALM. The E-node receives input only from the A-node of the CALM to which it is attached and sends random activations to all the R-nodes of the CALM. These random pulses fall within the range $[0, a_E(t)]$, where $a_E(t)$ stands for the activation of the E-node at the time t. What the E-node does by this construction is basically rattling the R-nodes. If several R-nodes within the CALM are highly active—when the CALM is not able to find the right category for the input—this E-node activation gives all the R-nodes a different random stimulation. Since these stimulations vary, one of the competing nodes will probably get a larger extra stimulation than the other ones, which results in its activation going up further than the activation of the other nodes. This solves the competition within the CALM (not immediately, but in the long run the advantage given by the stronger stimulation will leave the node on top). Due to the continuous activation of the A-node, this external stimulation process will go on until the competition within the CALM is resolved. Figure 5.2 shows a CALM with 3 R-nodes (resulting in three V-nodes).

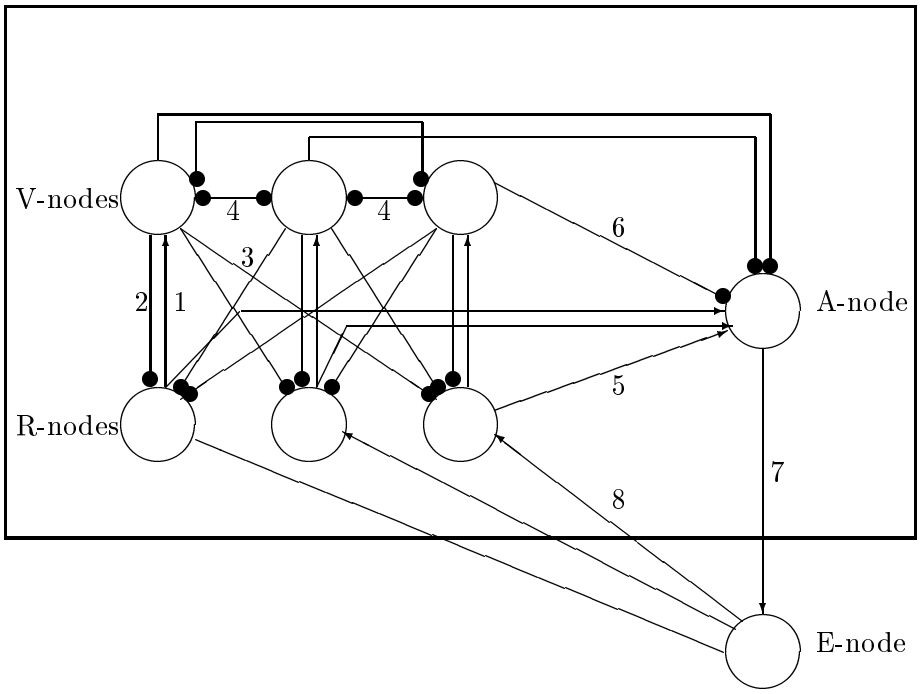


Figure 5.2: A schematic drawing of a CALM, with three R-nodes. It also contains all types of connections, numbered for type. For clarity's sake not all connections were numbered. The meaning of the numbers is:

1. Up-weight
2. Down-weight
3. Cross-weight
4. Flat-weight
5. Low-weight
6. High-weight
7. AE-weight
8. Strange-weight

5.3 The Internal Connections

Like the nodes, all the different types of connections have received different names. The weights of all these connections remain fixed at a freely chosen value during learning (the are determined before actually using the network, in fact, determining the parameter is closely related to determing the architecture of the network). The values of the weights connecting different CALMs all remain within the interval $[1, K]$. These weights will change during learning. Figure 5.2 also shows the types of connections possible in a CALM.

The different types of internal weights are:

- **Up-weight** (excitatory): connection from an R-node to its corresponding V-node.
- **Down-weight** (inhibitory): this is the counterpart of the up-weight connection. The up- and down-weights together cause an R-V-node pair to exhibit differentiating characteristics with respect to changes in activation input to the R-node.
- **Cross-weight** (inhibitory): control recurrent lateral inhibition from V-nodes to R-nodes. Normally these weights are significantly larger than the down-weights, to enhance the performance of their own paired R-node.
- **Flat-weight** (inhibitory): control the competition between the V-nodes, and thus between the R-V-node pairs.
- **Low-weight** (excitatory): control excitation of the A-node by the R-nodes. If a lot of competition occurs between the R-nodes, these connections get the activation of the A-node up, as to later resolve the competition between the R-nodes through the E-node.
- **High-weight** (inhibitory): control inhibition of the A-node by the V-nodes. These weights are relatively low compared to the low-weights, this to make sure the A-node does get activated if to much competition occurs, and to keep its activation relatively low when the competition is within boundaries.
- **AE-weight** (excitatory): regulates the activation of the E-node if the competition within the CALM can't be resolved.
- **Strange-weight** (excitatory): regulates the random activations used to resolve activation. In contrast to the other connections, the weights of the strange-weight connections differ within a module.

Appendix A sums up the initial values of the CALM parameters, including those of the weights.

As already suggested, all weights of the same type—strange-weights excepted—within one module are set to the same value. These weight values are parameters to a certain CALM, and do not change during learning. Since we want the connections to perform certain functions within a CALM, we can say something about the values of certain weights relative to others. The cross-weights for instance are meant to increase the chances of the R-V-node pair, from whose V-node the cross-weights leave. Therefore the value of these weights has to be far higher than that of the down-weight. In this way the activation of the other R-nodes is pushed down far more strongly than the R-node of the R-V-node pair, thus increasing its chance of winning the competition. Another example is the ratio of high- and low-weights. The A-node has to be activated if the competition can't be resolved, but it should not be activated if there is little competition. In fact, it is important for the correct functioning of the module that the activation of the A-node decreases to zero as soon the competition is resolved. To make sure this happens, the high-weight has to be higher than the low-weight.

5.4 The External Connections: Learning

There is just one kind of weight between interconnected CALMs: the *inter-weight*. Connected CALMs are always fully connected, that is if CALM 1 receives input from CALM 2, every R-node in CALM 1 receives input from every R-node in CALM 2. Connections between CALMs are normally directed, but can be chosen to be reciprocal. All inter-weights fall within the interval $[0, K]$, with K one of the learning parameters of the learning rule. The weights are adjusted using an adaptation of Grossberg's learning rule, which is related to the Hebb-rule during learning. In the Hebb-rule the weight depends on the correlation of the activations of the two connected nodes: if they have similar activations the weight is increased, if they differ the weight is decreased. It should be noted that changes in weight only appear when at least one node is active (during learning only external weights, that is, weights of connections between CALMs, not within CALMs, change). Grossberg [Gros 76] modified this by also taking the total background excitation into account. This background excitation is caused by the neighbouring R-nodes which give activation to the corresponding R-node in the other module (since two connected CALMs are always fully connected). Another adaptation is that the weights can only asymptotically approach their minimum and maximum values.

Two adaptations were made to Grossberg's scheme: the learning parameter μ was made dependent on the novelty of the presented input pattern. Secondly instead of using the unweighted background activation, $\sum a_f$, the weighted sum, $\sum w_{if}(t)a_f$, is used. The w_{if} are the weights between node i and all the nodes f it is connected to.

This immediately takes care of the awkward question of how a node can get its background activation without going through the weighted connection. This adaptation thus enhances biological plausibility.

All this results in the following learning rule:

$$\Delta w_{ij}(t+1) = \mu_t a_i[(K - w_{ij}(t))a_j - L w_{ij}(t) \sum_{f \neq j} w_{if}(t) a_f]$$

where a_f, a_i and a_j stand for $a_f(t)$, $a_i(t)$ and $a_j(t)$, respectively. $w_{ij}(t)$ is the inter-weight from R-node j to i (where i and j are R-nodes in two different, connected CALMs), and the $w_{if}(t)$ denote the factor representing the aforementioned background activation. $\Delta w_{ij}(t+1)$ denotes the change in value of the weight of the connection from R-node j to R-node i. L and K are positive constants, f, i and j are node numbers. The $K - w_{ij}$ term is always positive, since K is the upper boundary of w_{ij}, it also helps inflicting the upper boundary K. The second term within brackets lets the $\Delta w_{ij}(t+1)$ go down further proportional to the amount of background excitation. A high background excitation combined with a high $w_{ij}(t)$ might even make the weight go down, even though a_i and a_j are correlated. In this manner an overall down scaling of the weights occurs, thus preventing the weights from suffering from ceiling effects. The importance of preventing these ceiling effects have been discussed by Carpenter and Grossberg [Carp 87] within the framework of their "Weber Law rule". It should be noted that the nodes adding to the background activation do not lie in the same module as node j.

If node j is inactive, the weight will decrease, since only the second part of the formula between brackets is significant. μ_t represents the Hebb-parameter, controlling the learning rate of the module, and is defined as:

$$\mu_t = d + w_{\mu E} a_E$$

where d is a low valued constant, and a_E is the activation of the E-node. To make sure $w_{ij}(t+1)$ stays within the interval $[0, K]$ it is calculated in the following way:

$$w_{ij}(t+1) = \max\{\min[w_{ij}(t) + \Delta w_{ij}(t+1), K], 0\}.$$

It should be noted that this adaptation is only made for safety reasons, since the values of w_{ij} will stay within the interval for normal—that is low enough—values for μ_t. This implies that all connections between CALMs are excitatory.

Due to the changes of the inter-weights, each of the input nodes get has certain R-nodes with which it is connected stronger than to others, which is exactly what induces the CALM to categorize one input to one R-node and another to a different R-node.

5.5 CALM's Functioning

There are three basic processes that determine the working of CALM: *excitatory, inhibitory* and *arousal* processes.

The excitatory stimulation comes from other modules (within the network), the E-node or receptor nodes (in this case represented by input patterns). Initially all modifiable connections have the same weight (since an input pattern in the used implementation is a special case of a CALM, this simply means all connections between CALMs), resulting in all R-nodes within a single CALM being activated equally. After this all V-nodes get activated. The total process results in a lot of competition, causing the A-node to be activated, which triggers the random activation process by activating the E-node. In short this means that the excitatory processes trigger the total working of the CALM system.

The inhibitory process starts as soon as the V-nodes get activated and start to inhibit R-nodes, A-node and the remaining V-nodes. The process of mutual inhibition creates competition between the V-nodes (through the flat-weight connections) and starts, because of the ongoing activation of the R-nodes, an oscillating process in the activation of the V-nodes. It should be noted that this oscillation only occurs when there is real competition and not when the input is easily categorized, in fact the oscillations can be seen as an indicator of competition. The inhibition of the R-nodes can be split in two kinds: the inhibition of the R-node with which the V-node is paired up (through the down-weight connections), which is merely to keep activations within certain boundaries (it can even be kept at zero for networks consisting of a single CALM), and the lateral (horizontal cross) inhibition of the R-nodes (through the cross-weight connections), which is far more strong than the inhibition of the paired R-node. The intensity of the competition is regulated by the flat-weight and cross-weight connections. The higher these connections are, the stronger activations of other nodes are vetoed. This results in faster categorization if one R-node repeatedly receives more input than the others. If there are bidirectional connections between CALMs, the down-weight connections can be used to prevent reverberative action between the winning R-nodes in the CALMs by giving them a small value, after that, they also take care of the decay of activations of previous inputs.

The function of the arousal process is twofold. First it is meant to resolve competition between several similarly highly activated R-nodes. Since high competition results in an activated A-node which triggers the E-node, the E-node starts to give of random activations to the R-nodes, which will give one R-node the advantage that makes it win the competition (eventually). The second function is to enhance learning. When we learn the network a certain pattern, we want it to recognize the pattern not only when it is exactly the same as the one it originally learned, but also when it is slightly damaged (one of the strengths of neural networks is this kind of generalization. This feature is also found in humans). When we add some random noise

(which is what we do through the E-nodes), only a categorization that can survive slight adaptations in the pattern will be able to survive. Due to the functioning of the E-node, this random noise is much stronger for novel patterns—since they cause a higher amount of competition—than it is for known patterns. This causes the network to learn more when it is still unfamiliar with the pattern, than it does when it has already been offered the pattern before. Since we want the A-node to stay quiet when just one R-V-node pair is active, the low-weight connection has to be higher than the high-weight connection. However, when several R-nodes are activated, we want the A-node to become active. This is caused by the flat-weight connections: when just one V-node is active, these have no function, it only inhibits nodes that are inactive anyway; when several V-nodes are active, though, it takes down the total activation of the V-nodes, thus making the influence of the R-nodes greater. This results in the A-node only being activated when several R-nodes are active. Due to the initial activation of the V-nodes the activations of the R-nodes are pushed down, causing the activation of the R-nodes to go up again, etcetera. This oscillatory process occurs, as mentioned before, when there's a lot of competition in the module, but will eventually cease due to the advantage one of the R-nodes—the ultimate winner—gets through the random activation from the E-node. As the competition decreases, the activation of the winning R-V-node-pair increases, until it finally reaches a stable situation, with all the other R- and V-nodes unactivated. When this state occurs we say the CALM *converged*. Convergence is most often used as criterion to stop presenting the pattern, at least for the output CALM: it is possible for the output CALM to reach convergence without all CALMs in the network having converged. The winning R-node will remain activated as long as the input pattern is offered, but its activation will deteriorate when the pattern is taken away. The coding of the pattern, however, will remain in the changed inter-weights.

On the average one can say that there's a trade off between strong oscillation and changes in the inter-weights. When all the weights are still at initialization value, changes will be most strong: all R-nodes will receive the same activation. Since not all of the input patterns nodes are activated, the learning rule will strongly diminish the values of the weights between those input pattern nodes that are not activated, and the R-nodes of the CALM [1]. This change in weights, however, is not the real learning yet, since all input pattern's nodes weights receive the same value. The real learning starts when the CALM starts to categorize the input to one of the R-nodes of the CALM: the weights between this particular R-node and the activated nodes of the input pattern will become stronger than other weights. The more input patterns have been coded in a CALM, the more the weights will have changed, thus resulting in less oscillation in the CALM, after all, the most important cause of the strong

[1]It should be noted that the input pattern can be external, a real input pattern, as well as the R-nodes of a different CALM. Which of these is the case is of no consequence for the CALM, since the input pattern is implemented as a stripped CALM

oscillation is the fact that any R-node could be the R-node to which a certain input pattern is categorized early on, which is not the case anymore when the CALM has done a fair share of learning. Re-offering a CALM an input pattern it has already learned, generally leads to quicker convergence of the module, and firmer coding of the pattern.

5.6 Categorization and Learning

One of the features of CALMs is that they can learn without assistance: they are very suitable for unsupervised learning. This feature emerges from the fact that CALM is a categorizing system, as opposed to associative systems, such as BPN (which can only be used with supervised learning). When a CALM has learned to categorize a certain pattern, it has converged to one node, which is then said to represent the input pattern in the node. Figure 5.3 (adapted from [Murr 92]) shows an example of learning in a two R-node CALM. Early on all the inter-weight connections are equal, since the CALM has not learned anything yet. Since all inter-weights are the same, both R-nodes get activated equally (iteration 2), which gets the A-node activated (iteration 3), and in itself activated the E-node (iteration 4). In iteration 5 one can see that the right R-node got the advantage of the random activations that both R-nodes got because of the arousal of the E-node, resulting in a strengthening of the inter-weight from the left pattern node (the one that is activated), and the right R-node (the R-node that has become the category of the offered pattern). The weights of the inter-weight connection from the second pattern node to the second R-node drop significantly, since the R-node is activated, while the pattern node is not. At iteration 21 the activation of the left R-node has dropped to nil, while the right R-node is still strongly active, as is the right V-node. Note that when both R-V-node pairs are activated the lateral inhibition of the V-nodes makes sure that the A-node is activated (we want this to happen, since there is strong competition between nodes), while it has lost its activation when competition has been resolved. In iteration 21, when the pattern has already been learned, the E-node is still activated, exposing the CALM to noise, thus enhancing a more stable coding of the input.

5.7 Supervised Learning

As mentioned in the previous section, CALM is a network paradigm that normally learns with unsupervised learning. However, unsupervised learning only works for a certain class of problems: those where the input patterns have emergent features, features that separate them from the other cases, that the network can pick up on.

If, however, we want a CALM to learn the categorization of patterns into categories defined by us, we need to be able to tell CALM which category the pattern should

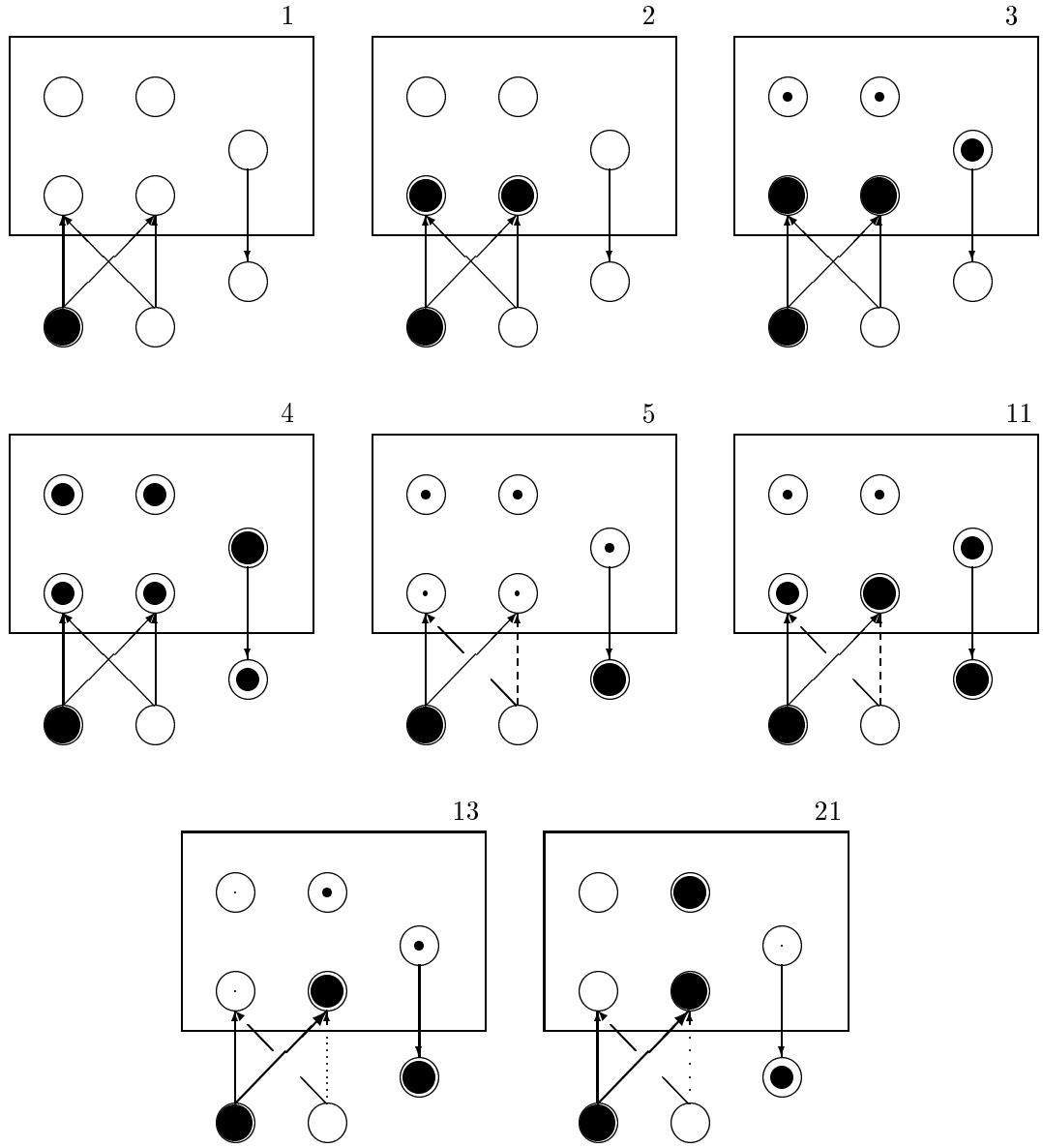


Figure 5.3: An example of learning in a CALM with 2 R-nodes, the numbers at the upper left of each module indicate the number of the iteration, the size of the dot in the nodes gives the amount of activation, the thickness of the arrows the size of the weight. For clarity, the connections other than the inter- and AE-weight connections were omitted, since these are of no importance to learning.

be put in.

There are basically three methods to achieve this CALM variation on supervised learning (that is what we are doing). The first method is to give the V-node paired up with the R-node to which we want the CALM to converge a starting activation. This starting activation works as a strong impairment on the other R-nodes, thus giving the desired R-node such a strong advantage over the other R-nodes, that the CALM converges to this R-node (assuming we chose the starting activation in the right interval, of course). Giving the R-node an initial activation would be less effective, since all R-nodes are immediately activated by the input nodes, which results in the pre-activated node soon losing it's advantage. Activating the V-node, however, means that the activations of all R-nodes to which the V-node is not coupled are kept down.

The second way is to present the CALM with two input patterns: one the original input pattern, the second the desired output pattern. This second pattern exists of exactly as many nodes as the output CALM of the network does, with one activated R-node, representing the category of the input offered on the input pattern. This hints the network which inputs are to be put in the same category, and which are to be put in different categories. The network, however, is free to choose on which of the output nodes it represents the category, that is the output CALM not necessarily converges to the node activated in the supervise pattern, but there is a one-on-one correspondence between one node in the supervise pattern and one node in the output CALM (assuming the network has properly categorized the inputs. This is not something that always happens: when the parameters are too far off "good" values for the network, the network will perform poorly, not only resulting in bad categorizations, but also in ambiguous categorizations). After learning the network a problem, we have to find what nodes in the output CALM the categories have been mapped upon. To achieve this we offer the network the supervise pattern only, the input pattern is left unactivated. This way the output CALM converges upon the supervise pattern only, thus indicating on which output CALM node the category is mapped. This second method is far more biologically plausible than the first: when we learn we do not have one neuron, or group of neurons, previously activated to enhance learning, we simply get offered the problem and the solution: two inputs, one the problem, the other the way to learn it. This is very similar to the second supervised learning method described.

A third method is an adaptation from the second method. In stead of the normal full connection between patterns and/or modules, a one-to-one connection is chosen between the supervise pattern and the output CALM. This means that there is a one-to-one connection between a node in the supervise pattern and the output CALM. Unfortunately, the implementation of this method is rather cumbersome within the environment used for this research: all functions are built to work with fully connected patterns and/or modules, thus making it necessary to change a lot of functions. This is caused by the node connections being implemented as module connections. Since

all patterns are implemented as special cases of modules, pattern-module connections are full connections automatically.

The third method was not used in this research.

5.8 Convergence in CALM-networks

The normal convergence criterion for a CALM-network is, as mentioned before, a unique categorization for the offered pattern. This strategy, however, limits a CALM-network in its "expressive" capabilities: systems of coarse coding (the spreading of information over various nodes, which generally means a potentially far larger chunk of information can be coded with a certain number of nodes) are excluded by it. In this research, as most often, this unique categorization criterium is used solely for the output-CALM, thus making is possible for the internal modules to still use coarse coding. It might be interesting to do some research in the future after coarse coding in the output module, even though this makes the interpretation of the output as well as the learning of the network theoretically more complex.

Normally, a module is said to have converged when one node is above the high cenvergence criterium, while all the others are below. See Appendix A for default values.

5.9 Bias-nodes

A common flaw of Artificial Neural Networks is that they are hardly able to categorize low input, that is, input patterns that hardly bring any activation to the network, yet we may want this kind of input categorized anyway. A common, and effective, way to solve this problem is by adding a so-called bias node to the input pattern. This bias node has the same activation for all patterns, it therefore does not give any information, it does, however, result in a certain activation. This way we get the network activated, whether the input pattern itself brings activation, or not.

A further advantage of this method is that the network seems to gain in calculative power: when using n nodes for an n-wide pattern, the hyperplanes always go through the origin of the problem space, when using an $n + 1$-wide pattern, the hyper planes gain freedom, thus making for wider possibilities in categorizations. Examples of this can be seen in some of the pictures in chapter 10.

Chapter 6

Biological Plausibility

6.1 Modularity in Nature: the Modularized Brain

One of the simplest definitions of *modularity* is *a subdivision in identifiable parts, each with its own purpose or function* [Boer 92]. Using this definition almost everything has some kind of modularity, no matter how rudimentary it is.

Just like almost everything else, a being, be it human or not, can be split into modules. One of the most obvious is a decomposition in organs, which can be decomposed in cells, which can be split in the organelles that are in the nucleus and the cytoplasm. The different organelles have different functions within in their cells, as do the cells in their organs, as do the organs in the total organism.

As all organs, the brain can be decomposed in structures, which make the brain into a strongly modularized whole, each module consisting of a number of neurons.

Two of the most prominent structural characteristics of the brain will be referred to as *horizontal* and *vertical* structure [Bloo 88]. Horizontal structure is found where processing in the brain is carried out by subsequent hierarchical neuron layers. Such multi stage information processing can for example be identified in the primary visual system, where simple features of visual images as line and arcs are represented in layers of *simple neurons* which are combined and represented by neurons in subsequent layers that possess increasingly complex representational properties (e.g. [Hube 62]). The vertical structure of the brain allows for separate processing of different kinds of information. An example for this, too, can be found in the visual system: *form, colour, motion* and *place* are processed in parallel by anatomically separate neural systems. This information is integrated at higher levels by convergent structures. The presence of horizontal and vertical structures makes the organization of the brain modular, as does the subdivision of the brain into two hemispheres. This subdivision does constitute a hemispheric specialization. Striking examples of this can be seen in so called *split brain* patients, patients who had their *corpus callosum*, a large structure that connects the two hemispheres, severed. These patients can function normally

up to a high degree, since the communication between the two hemispheres can go through the outside world. Experiments with such patients show that there is a large modularization, of even the most complex functions [Kand 85].

More evidence for this functional modularization stems from psychological *interference* experiments. Subjects offered totally different tasks at the same time are very capable of completing these without problems. When we, for instance, walk through the woods, we have to do a lot of things at the same time: listen to the birds, stay on the path, watch nature and make sure we don't trip over a root or walk into a protruding branch. Apart from that, subjects also do rather poorly in tasks that are very similar, for instance listening to different auditive inputs, on each ear ([Allp 80]).

The localization of functions in the brain and its modular structure speak in favour of the functionality of the modularization: the modularization of the brain enhances its functioning.

The question is when this modularity arises. The anatomical location of certain functional centers and the fact that they are in the same place in most persons, speaks in favour of a certain predefinition, which would have to be genetic. Another argument is the fact that development of most children, for instance linguistic development, follows the same way, no matter where they develop. Since the genes do not have enough informational capacity to code every single connection of neurons, the coding of the connections has to be modular.

6.2 Modularity in Genes

The genetic coding of all life forms on earth is also modular. All genetic information is stored in genes, which are contained in long double stranded helical strings, DNA, of which each cell in an organism has a copy. This genetic information is a digital coding with four different bases: *adenine, guanine, thymine* and *cytosine*. The information stored in the DNA helix is transcribed into strings of RNA, which is an inverse copy of a part of the DNA. RNA is built from the same bases as DNA, except thymine which is replaced by *uracil*. Each triplet of bases in the RNA forms a coding for one of 20 amino acids, or a *marker*. The RNA is translated into proteins by a *ribosome* that reads the RNA, and connects the amino acids in the order coded in the RNA. Markers tell where to start and where to stop reading the RNA string.

Amino acids are the building blocks of which all proteins are built. Most of the proteins are *enzymes* that catalyze the different chemical reactions in the cells. Each protein consists of a sometimes very large number of amino acids put together in a specific order. It is this order that determines the shape, and through that shape the functioning of the protein. For each protein, of which about 30,000 exist in humans, the order of the amino acids is written in the DNA, where each protein is coded by one gene. In this way, the DNA determines what kind of proteins are built and therefore how *differentiation* takes place. This differentiation is caused by the forming of certain

proteins that generate a positive feedback on the genes in the DNA that produced these proteins. These proteins will also repress another group of genes which will never be active again. Embryo-logic experiments show also that certain cells in an embryo control the differentiation of adjacent cells [Guyt 86], hereby implementing the idea of *context*. Most mature cells in humans only produce about 8,000 to 10,000 proteins rather than the total amount of 30,000. It is this process of cell differentiation that determines the final shape of the organism in all its details.

At birth the brain is supposed to have an initial structure. It is the process of cell differentiation that forms the shape of the brain in a way that is still unknown. But somehow the initial structure has to be coded in the genes (genetic modules of information, or building blocks).

6.3 Artificial Modularity

After reaching the conclusion that modularity might play an important role in the functioning of our brain, we might wonder if modularity would work well in ANNs as well. So far results appear to be pointing that way ([Murr 92], [Happ 92] and [Norr 90]), but since managing large ANNs is still a problem, only research on smaller networks has been done. Another problem is that even for networks with just a couple of layers, the mathematical equations become already so complex that is practically impossible to still know what happens within the network: since it is impossible to calculate what kind of network topology works well, we have to come up with one and then test whether it works. From the testing point of view this scheme is certainly feasible, so all we need now is an attractive, and preferably effective, way to move through the space of possible topologies. As we used reverse engineering to move away from the not so well working algorithmic solutions we turned to ANNs, so why not try this scheme again: use a metaphor for the natural development of neural networks. This leads to the use of genetic algorithms (GAs). Since we know how to test a network, and have a good idea of what we want in the network, we can use these ideas in developing a good fitness criterion, with which the GA can be provided, in that evolving to a better, hopefully optimal, solution.

Now that we have come this far though, we still have not solved all problems: a gap remains between the GA and the ANN. Even though we know how to communicate from the ANN to the GA how well we consider a network to be, we still do not have a method for the ANN to understand what network the GA has coded: an interpretation of the string is missing.

An ANN can be seen as a graph, so what we could use is a method to generate graphs. Artistid Lindenmayer provided us with one with his L-systems ([Lind 68]), a method he used to describe the development of multicellular organisms. A good side of L-systems is, that they are, as ANNs and GAs, biological metaphors, which made them very suitable for this type of research (the original idea stems from [Happ 92]).

Another advantage of L-systems is that they are suitable for both the development of vertically and horizontally organized networks, apart from the fact that they can use context as well (which—as mentioned before—can also be found in the development of brain structures). It should be noted that L-systems are suitable for graph development, but were not specifically developed for it. The biological plausibility of L-systems in this context is disputable. As mentioned before, it is not completely clear yet how exactly the human brain is guided in its development by the genes. However, since L-systems are a biological metaphor, we stay as close to nature as is possible with current knowledge.

The method used in [Boer 92] can be summarized as follows:

1. A genetic algorithm generates a bit string, which is the chromosome of a member of its population. The search of the genetic algorithm is directed towards a member with a high fitness, a measure resulting from step 3.

2. An L-system implements the growth of the neural network that results from the recipe coded in the chromosome. The chromosome is decoded and transformed into a set of production rules. These are applied to an axiom for a number of iterations and the resulting string is transformed into a structural specification for a network.

3. A neural network simulator (for instance BPN or CALM) trains the resulting network structure for the specified problem. The performance of the network is in some way expressed in a fitness, a measure for the networks performance, which is returned to the genetic algorithm.

In this research several methods were used, of which the previous is one.

6.4 CALM

At first sight one would say that CALM is far more biologically plausible than for instance BPNs are.

CALM uses a Hebb-type learning method, which is quite similar to the way learning takes place in the human brain: weights are changed based on the correlations between the activations of the two nodes connected by the connection. This is far more biologically plausible than the Back Propagation of an error through a network, like Back Propagation uses.

Besides this, a lot of network paradigms use connections that can simply change sign, with the connection simply changing between excitatory and inhibitory behaviour, while in the human brain a neuron either inhibits or excites another neuron,

without ever changing. All connections within a CALM are either excitatory or inhibitory, and remain so, while the connections between CALMs always are positive. This makes CALM highly biologically plausible as to connections.

A third argument for CALMs biological plausibility is the fact that CALMs are already structured of themselves. Networks like BPNs simply exist of a number of loose nodes that are interconnected; even though structure can be imposed on them, they are basically unstructured. CALM networks, on the other side, have a twofold structure: the tight organization of the CALMs, which could in a way be compared to the mini column as found in our brains [Murr 92], within a network upon which, like on a BPN, a structure can be imposed.

At second sight, however, it appears that the CALM-mini column analogy doesn't fully hold. One of the types of cells in a mini column are the *pyramidal cells*, which appear to have a functional resemblance with the R-nodes of a CALM ([Murr 92]). However, a mini column has some 60% pyramidal cells, which, beside the fact that this fraction is significantly higher than the fraction of R-nodes within a CALM, seems to contradict the strict pairing of R- and V-nodes. To overcome this deficiency, some changes to the CALM paradigm have been proposed ([Murr 92]).

The first is to let the number of V-nodes in a CALM be a lot smaller than the number of R-nodes, with it dropping the strict pairing of V-nodes to just one R-node. In fact, every R-node is connected to several V-nodes. Using this construction, the flat-weight connections become superfluous, since the concept of lateral inhibition has lost its meaning with every V-node receiving input from several R-nodes. The connecting of an R-node to several V-nodes also results in a change of the cross-weight connections: since there are less V-nodes than there are R-nodes, a V-node receives input from more than one R-node. Only the R-nodes from which a V-node does not receive input are connected to it with a cross-weight connection. Despite this change to the internal wiring of a CALM, the functioning of the network (that is, the competition and the arousal systems) remains the same, due to the *indirect competition* triggered by the cross-weight connections. To make sure indirect competition will result in the network to converge, two different R-nodes can never be connected to the same set of V-nodes. If this constraint weren't inflicted, the possibility would exist that competition could not always be resolved: if the set of V-nodes to which a particular R-node is connected is the same as that of another R-node (all R-nodes are connected to the same number of V-nodes), competition would not be resolved, since the nodes would receive equal inhibition. Indirect inhibition does not result in every R-node inhibiting the other R-nodes equally, but so far tests have shown similar categorization results for this adapted CALM and the original, although the fact that the interconnection of R- and V-nodes is constrained, has some influence on the initial categorization of patterns (that is, the node chosen for the first pattern of a certain category). If for instance the patterns (0,1) and (1,0) were offered, they would be mapped to nodes that are in maximal indirect competition (that is, have

the smallest intersection of connected V-nodes possible). This however, is merely an internal change, but has no relevance for the outside: if patterns were in the same category before, they still are. For a more elaborate treatment of this adaptation see [Murr 92].

Another adaptation that enhances biological plausibility is to put all the E-nodes of all the CALMs within a network together in one structure. Global excitation could in this way stimulate elaboration learning (the deeper learning of previously encountered patterns) through the whole system. Studies have shown ([Murr 92]) that lesioning the E-node structure (damaging it), results in the system losing its ability for elaborate learning. The same kind of behaviour is shown in sufferers of *anterogade amnesia*, loss of memories from before the occurrence of a lesion [Warr 70]. Since this kind of behaviour often occurs in patients with lesions to the hippocampus and amygdala, there is some reason to suspect the functioning of CALMs being similar to the functioning of these structures.

A final, and somewhat discouraging, thing on CALM's biological plausibility (as well as other ANN's, but to a lesser degree, since they are less complicated) is that we really do not know what happens inside a CALM network, just like we do not know what happens inside the human brain. In this aspect ANNs have failed: we have not been able, so far, to build an ANN which is a simplification of the brain and that we do understand, with it giving us more insight in the working of the brain. Of course, from a more computational point of view, ANNs have proved to be very succesful: they are at the moment use in a wide variety of fields, often working on problems on which traditional algorithmic programs performed rather poorly.

Chapter 7

Implementing Genetic CALM Structuring

The software written and used in this research consists of several more or less independent libraries and programs. The CALM part is represented by CALMlib, a library of CALM and CALM-associated functions mainly written by Bart Happel, Jaap Murre and Nico Mulder, more complete treatises than the one given here can be found in their work ([Happ 92] and [Murr 92]). The genetic part is represented by `CalmGenalg` and `CalmExtGen`, adaptations from `GenAlg` and `ExtGen` (a genetic library) which were developed for coding BPN architectures, to the CALM paradigm. `GenAlg` and `ExtGen` were taken from the software used by Boers and Kuiper on their research ([Boer 92]). The production rules are rewritten using the program `CalmLSystem`, adapted from `LSystem` (again, a BPN predecessor from Boers' and Kuiper's work). The `LSystem` program transforms the axiom to the final string by using the production rules resulting from the genetic manipulating (this process will be explained in the following section). It finally writes a matrix which represents the architecture of the network, which is read and interpreted by CalmFunc, the module that creates the CALM-network and tests it. All the software is written in C.

7.1 The Grammar

In this research a context sensitive grammar is used to create CALM networks. The strings used in this research are composed of letters from the alphabet {A-Z, 1-9, [,]} $\cup$ {,} [1]. An R-node is represented by a letter. If two letters are separated by a comma, no connection between them is made (this means that they will end up in different

[1] It should be noted that this system is the one originally used in the research by Boers and Kuiper ([Boer 92]). Since then Boers has adapted this scheme. In this new notation all skips numbers are upped one, and the , is replaced by 1 ([Boer 95]).

CALMs, since no R-nodes within a CALM are connected), if they are adjoining no connection is made. Modules are created by grouping a number of characters by using the [and]. All *output nodes*, nodes that have no output to other nodes in the module, of a module are connected to the *input nodes*, nodes that receive no input from other nodes in the module, of any module it is connected to. These modules can be seen as networks connected to form one greater network. Like with characters, adjoining modules are connected, while those separated by a comma are not connected. This type of connection only provides a rather feed forward oriented type of networks. To enhance more complex types of networks, *skips* were introduced. Every single number in a string denotes a skip. A digit x means that the next x node units should be skipped, where a node unit can be a single node or a full module. This means that, for instance, in the string $A1BC$ A is connected to C and that in the string $A1[BC]D$ A is connected to D, and not to C. If a skip exceeds a module, it is continued after the end of the module as can be seen in the example in figure 7.1. This figure shows the Back Propagation network that would be generated by the string [A2[B,C]D]E, using the interpretation that was used by Boers and Kuiper ([Boer 92]). It should be noted that all connections were chosen to be forward. Figure 7.2 shows the CALM network generated by the same string. The lowest level of modules in the net (nodes with the same input and output are put in one module) form one CALM. However, since these modules can often contain just one node (as can be seen in the network of figure 7.1), and categorization by one category is rather useless, the number of nodes is increased with one, by this making sure each CALM has at least two R-nodes. Apart from that, all connections are chosen to be reciprocative.

7.1.1 Production Rules

As mentioned in chapter 3, the production rules of the context sensitive grammar are of the form:

$$L < P > R \rightarrow S$$

with the characters taken from {A-Z, 1-9, [,]} $\cup$ {,}.

Predecessor

The predessor (P) may only contain complete modules and nodes. Therefore the number of left and right brackets must be equal in a correct order. Each module must be complete. This means that the string A]BD[D (for example) will be discarded. The predecessor may not contain empty modules ([]) and should contain at least one letter (node).

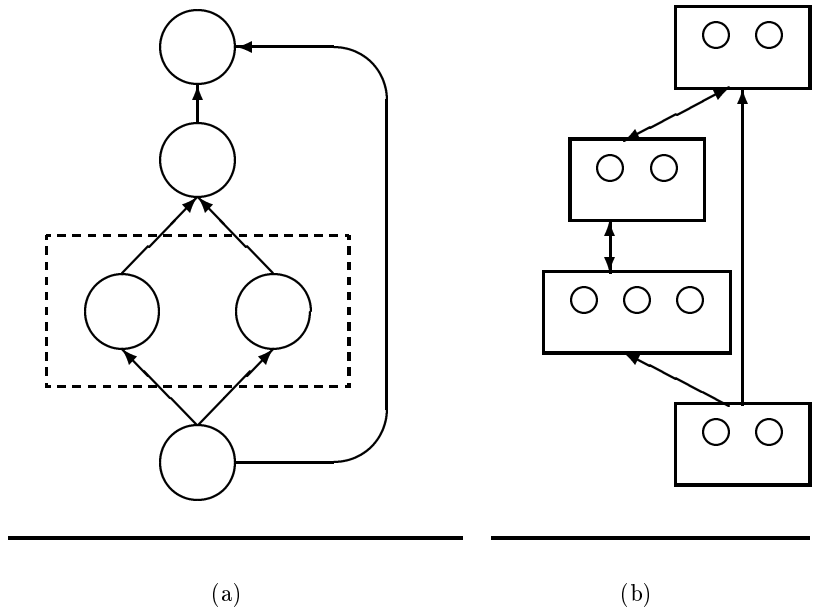


Figure 7.1: The network generated by $[A2[B, C]D]E$ (a) and a simplified version of its CALM counterpart (b) (only R-nodes were drawn). Note that each module from (a) has received an extra node, this to prevent categorization with just one possible category. All connections between CALMs are chosen to be reciprocative.

Successor

The successor (S) has the same constraints as the predecessor. The successor may be absent, in which case the predecessor is removed from the original string when applying the production rule.

Context

If a context is present, the same constraints as for the successor and predecessor apply. In addition, no loose digits are allowed: each digit must follow a node or module. For example, the string 1A[B] is not allowed because of the leading 1.

A deviation from the L-systems described in chapter 3 is the handling of context. Whereas in those L-systems the context is matched against the characters adjoining the predecessor (left and right of the predecessor for the left and right context) in these L-systems the context is formed by the nodes that give input (left context) and those that receive input (right context).

Because of this special interpretation of context, the context string should be seen as an enumeration of a number of nodes and/or modules, all of which should be connected to the predecessor at the time of context matching. The context must be a subset of all nodes connected to the predecessor (left context) or all nodes the predecessor is connected to (right context) in order to match.

Let us look at an example with the following rules:

1:			A			→	BBB
2:			B	>	B	→	[C,D]
3:			B			→	C
4:	C	<	D			→	C
5:			D	>	D	→	C1

We take A (see figure 7.2a) as axiom, which means that in the first step only rule 1 applies, resulting in the string BBB (7.2b). In the second rewriting step rule 2 matches the first B in the string, which is rewritten to CD. During the same step the second B is also rewritten using this rule (7.2d), as the third B is rewritten using rule 3 (7.2e). For the first and the second B, both rule 2 and rule 3 match, however rule 2 is chosen because it poses a greater constraint on the structure. All this results in the string [C,D][C,D]C after the second rewriting step. It should be noted that the string BBB is rewritten to the string [C,D][C,D]C in one step, the intermediate strings, of which the networks the present are shown in 7.3c and 7.3d, do not exist at any time, they are just added to enhance clarity. In the third rewriting step, rule 5 matches on the first D of the string. As already explained, this match isn't a textual one (the two Ds are not adjoining in the string), but a connectionwise match: the two Ds are connected in the network. The second D matches rule 4. This all results in the final

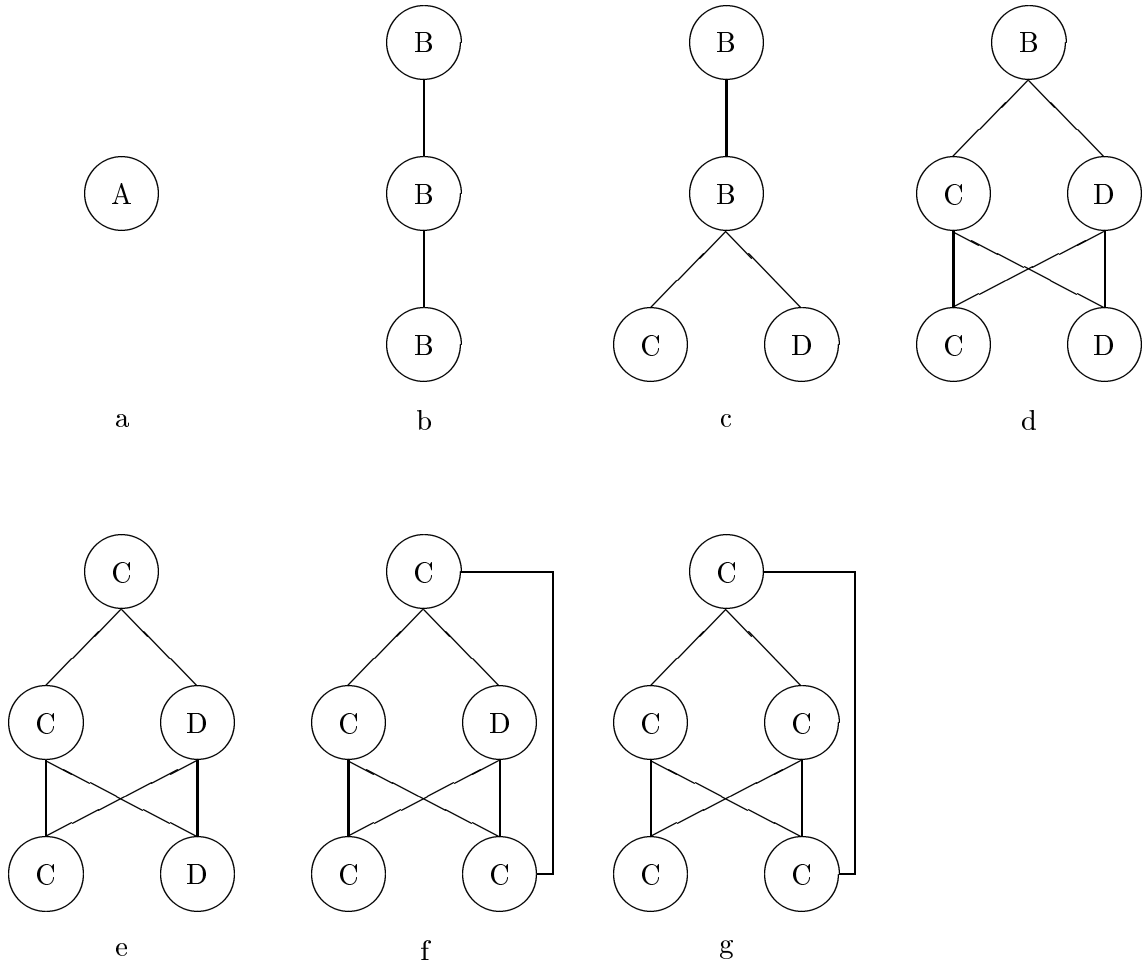


Figure 7.2: The rewriting of a network with one node A in it.

string [C,C1][C,C]C, which is shown in figure 7.2g. The 1 after the second C in the string denotes a skip of one node. As can be seen in the figure, it skips the complete second module ([C,C]), and not simply the first C in this module.

7.1.2 Coding of Production Rules

As described earlier, GAs work with coded parameters of a problem, and with the problem itself. This means that we will have to code the production rules we use in some way. Each production rule consists of four, possibly empty, parts. Each of these parts is a string over the alphabet {A-Z,1-9,[,]} $\cup$ {,}. This alphabet implies using

	00				01				10				11			
00	3	3	*	*	*	*	*	*	2	2	2	4	,	,	,	,
01	[	[	[	[	1	1	1	1	A	A	A	A	B	B	B	B
10	D	D	2	2	E	E	F	F	G	G	H	H	*	*	[	[
11	]	]	2	5	]	]	]	]	[	[	]	]	C	C	C	C
	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11

Figure 7.3: The coding of the symbols. To find a symbol, first find the correct row by using the first two bits, then the right quarter by using the bits on top, and finally the symbol by using the bits below.

26 letters, all representing nodes in the resulting network, however, any number of symbols representing nodes can be chosen. The same goes for the digits representing skips. In this research eight symbols for nodes (A-H) were used, and 5 symbols for skips (1-5). This results in an alphabet with 16 symbols (8 letters, 5 digits, 2 brackets and the comma). Each of the characters was represented by a bitstring of fixed length (l). As can be read from [Guyt 86], genetic coding is done in triplets of four possible different bases. Following this we would let each string exist of three triplets of two bits each (to code which of the four bases it is), giving us a stringlength of $l = 6$. The choice of this l leaves us with 64 possible bitstrings, each coding one of 17 symbols: the 16 symbols mentioned before and an extra symbol, the *, to denote the end of each string (this can be compared to the start and stop markers used for the transformation from RNA to protein ([Guyt 86]). The 17 possible characters have to spread in some way over the 64 possible codes. The table used can be seen in figure 7.3. Since connections were chosen to be reciprocative, there is no need for negative skips to be able to implement recursion.

The table is read as follows: to determine the character corresponding to a bitstring (of length 6), determine which of the four rows at the left corresponds with the first two bits of the string, then choose the right column from the second pairs of bits at the top of the table, and finally choose the right character using the pairs of bits at the end of the lines in the found 16th of the table. For instance 111000 corresponds to *: choose the lowest of the four 4x4 blocks, choose the third column and in that the first row.

In picking a production rule form a chromosome, the following method is used. Pick a starting point anywhere in the bitstring (chromosome). Start reading the L part (left context) of the production rule. This is done by reading six bits in a row, looking up its coding, and considering the left context read when six bits coding an * are found (since * is the separator). Start reading the P (predecessor), R (right context) and S (successor) parts in the same way. When all parts have been read, the reading of the next production rule starts, continuing until the string is exhausted. If

the end of the string is encountered before the production rule is finished, the partial production rule is discarded.

With real DNA, the strands are "read" by choosing a starting point within the bases on the strand and then reading triplets of bases until a punctuation mark is read (which can be compared with the asterisk), at which time the building of the current protein is "finished", and the building of a new protein starts. Research indicates that different proteins can be coded using the same genetic information, simply by using a different starting point and using different reading directions. Adapting this principle to our strings, we obtain twelve different ways of reading the strings: six different starting off-sets (starting six bits further down the string gives the same bases (characters) and two directions, starting on each of the 0th through the 5th bit from the start or the end, depending on the reading direction). Due to this high level of information in one string, the level of implicit parallelism might be significantly higher in this way than it is when reading a string just once.

Using chromosomes of length k results in $2{\cdot}6{\cdot}((k-1)div6)+(k-1)mod6$ characters. For chromosomes of 1024 bits long this results in a string of 2040 characters. The total number of possible valid production rules for strings of this length usually is about 250: the 64-characters translation table contains 8 asterisks, so a string of 2040 characters contains roughly $\frac{8}{64} \cdot 2040 \approx 255$ markers. Almost each marker can be used for a production rule (the last four markers placed at the end can not be used). If we discard all production rules that contain invalid character combinations (those described earlier), we are left with much less rules, of course.

7.1.3 Repair mechanism

With real DNA, in the few hours between DNA replication and mitosis (the actual process of a cell splitting into two new cells, [Bloo 88]), a period of very active repair and "proofreading" takes place. Special enzymes cut out defective areas and replace them with appropriate nucleotides. Because of this, mistakes in the process of cell replication are seldom made (these mistakes are called *mutation*). The software also contains functions to repair faulty strings. The functions remove spare brackets, commas, digits, etcetera in order for the string to meet the restrictions mentioned before. The number of production rules left after this process, is about 50.

7.2 CALMlib

CALMlib is a library of functions especially made for CALMs and CALM-networks. It includes functions for network creation, maintenance, input handling, showing the network and of course for calculating the activations of the nodes. In contrast with the most other software used for this research, the CALMlib was not adapted especially for it.

The most important functions in this library are:

- `calc_act`: calculates the activations in the next step and stores them in an intermediate variable.
- `calc_wt`: calculates the new learning weights, recursively if necessary.
- `swap_act`: swaps the new activations calculated with `calc_act` into the network.
- `net_def`: allocates memory for a network and creates the required structures.
- `net_free`: frees the memory used by a network.
- `mdl_def`: allocates memory for and creates a single CALM. Also connects the CALM to a network, as defined using `net_def`.
- `mdl_free`: frees the memory used by a single CALM.
- `ptn_def`: allocates memory for an input pattern. An input pattern is a stripped version of a CALM.
- `sall_act`: sets all activations in a module to a given value.
- `mdl_reset`: resets all activations to starting value. Used directly before offering a pattern.
- `mdl_con`: connects two modules.
- `mdl_tree`: shows a tree representation of a CALM or network.

More elaborate information on the CALMlib can be found in [Happ 92] and [Murr 92].

7.3 Extended `CALMGenLib`

The original extended `GenLib` only saw to the genetic creation of the structure of the network. For CALM, however, not only the architecture, but also the learning parameters were determined genetically. To provide for this, the definition of a member of the population was adapted to the following:

```
typedef struct
{
   float fitness;                          /* fitness of the member */
   unsigned *genPos;           /* pointer to array of gen positions */
   unsigned char **genValue;
```

```
          /* pointer to array of architecture chromosomes (strings) */
   unsigned char **parValue;
                       /* pointer to array of parameter chromosomes */
} MEMBER;
```

The pointer to the `parValue` array was added. This array contains 30 genetically coded parameters (as used by the `CALMlib`). The total population remains defined as:

```
typedef struct
{
   unsigned popSize,        /* number of members in this population */
            nrGenes,              /* number of genes in each member */
            genSize;        /* size of gene in bits (multiple of 8) */

   MEMBER   *member;
} POPULATION;
```

The `genSize` is limited to sizes multiples of 8, this to prevent the code from becoming unnecessarily complicated.

To allow for this change in the genes, most of the functions in the extended `GenLib` had to adapted:

- The creating functions (`DefinePopulation` and `DefineMember`) had to be adapted to also create an initial parameter population (this can be done by either using starting random values or by using the standard parameters).
- The disposing functions (`FreePopulation` and `FreeMember`) also had to dispose the structures created for the parameter chromosomes.
- The disk IO functions (`SavePopulation` and `LoadPopulation` also had to write respectively load the parameter chromosomes.
- The genetic operators had to be adapted, to also code the parameters. This formed the bulk of the adaptations to the `GenAlg`. At the moment the chance of each genetic operator is the same for parameters and architecture, this can be easily adapted, though.

The genetic operators used for this research are in the functions `BitCrossover, BitInvert, BitRankselect` and `Mutate`. The original extended `GenLib` contained a couple of more genetic operators, but these were intended for use with roulette wheel selection, which was not used in this research (in fact, these functions were not adapted to manipulate the population contain parameter genes).

7.4 L-systems

The main adaptation that had to be made to Boers' and Kuiper's L-systems, was the writing of the connection matrix. Their system uses simple nodes, so the matrix that was written as a nxn-matrix, where a one at position (i,j) denotes a connection from node j to node i. The modules in this system would form by themselves. However, for the CALMs not the lowest level nodes are connected, but the CALMs themselves are (as explained in an earlier chapter). This means that the matrix that is to be rewritten is not a connection matrix of nodes, but of connection matrix of CALMs. However, the loose nodes as found in the original matrix, are still of importance: they represent the number of R-nodes in the CALM they form. To prevent for this, an extra line was added to the matrix file. On this line the number of nodes (as determined from the string) in each module is written, As mentioned before, this number is incremented when creating the actual CALM in `CalmFunc`, to provide for CALMs that contain just a single R-node.

To reach this, a function was added to the L-system module. The program still creates the node connection matrix as was originally done, but this matrix is processed once more to transform it into a CALM connection matrix. This is done by simply taking a node, and finding all other nodes in the matrix that are connected to the same nodes, adding the number and noting.

The second adaptation is of less importance. In the original L-systems module, single node modules that were of no importance for the computational power of the network, but simply passed on information were shortcircuited: the connections made through them were made directly, pruning the superfluous node. However, in a CALM network such a single node would become a two node CALM, which does add to the computational power of the network, therefore the mentioned pruning mechanism was removed from the code.

7.5 `CalmFunc`

`CalmFunc` is the part of this software that was solely made for this research. Its function is to interpret the matrix as created by the L-system and transform it into a network. Apart from that it takes care of the reading, or otherwise offering of the input, as well as determining the fitness of the created network.

7.5.1 Creating the Network

The creation of the network is established using functions from the CALMlib. The network as created by the L-system is the internal part of the CALM, the part that does not communicate with the outside world. The input is of a set size: one pattern representing the input for the problem (as mentioned earlier a pattern is a stripped

form of a CALM) and a second pattern representing the desired activation for the output CALM. These two input patterns are connected solely to the CALMs in the network that are input CALMs: CALMs that receive no input (in the original structure created by the L-system) from other nodes. The output module is of the same size as the second input pattern, and is only connected to the output CALMs of the network (the nodes that did not give output in the original structure created by the L-system).

If two CALMs within the network are connected, their connections are reciprocative: they go both ways. This, of course, blurs the concept of input and output CALMs, which is why the original structure created by the L-system was referred to when speaking of input and output CALMs.

7.5.2 Input Problems

As a means of offering input to the network, several functions were written. Some of them are hard codings of problems (one function for instance codes the mapping problem, which will be explained in the next chapter). Apart from this some functions were written to read input from disk files.

7.5.3 The Fitness

One of the most important factors in the success of the GA is the fitness function chosen. If the fitness does not properly reflect what in a we are looking for in a solution, we can not expect the GA to find a satisfying solution.

There are three factors that are of basic importance for determining the quality of a neural network: the size of the network (number of nodes or modules), the speed of convergence (how fast the network comes up with a solution, preferably the right one) and finally how many of the offered problems were solved correctly by the network. This last factor is the most important one: for one thing we want the network to give as many good solutions as possible. For this research we gave the number of correct solutions to be of such importance, that a network can never have a lower fitness than a network that gives less correct answers. To be more concrete, we started out with a maximum fitness, in this case 1000 (since rank-based selection was used, the exact value was of no importance. It should be noted, though, that this number should be at least the number of offered patterns + 1 (to account for the number of modules in the network)), and for each wrong answer one was subtracted. Besides this, 0.01 was subtracted for each module the network had; since the network could not have more than 30 modules, the number of modules could not diminish the fitness by more than 0.3, thus making sure that the number of modules could not violate the aforementioned fitness constraint. The speed of convergence was not used, but can easily be brought into the fitness.

7.6 Using the Software

A number of different modules and programs have been written for this project. Since these are fairly stand alone, they are treated seperately here.

7.6.1 GA generated, L-system Assisted Architecture Creation

The unit of this program that starts everything is called `calmgenalg_w`. It takes a population and starts genetically manipulating this. It takes four parameters, of which the first three are mandatory. They are:

- **The simulation file**: this file contains some of the parameters used by the program. It could, for instance, look like this:

```
## Sample configuration file
##
#files          .
#population     test.pop
#control        test.ctl

#size           100

#pmut           0.005
#pcross         0.5
#sites          20
#pinv           0.3
#pressure       1.5

#steps          6
#axiom          A
```

  This sample contains all the recognized parameters of the parameter file. `files` determines the directory where all created files are put (the production rule, matrix and string of each chromosome are saved), `population` is the name of the population file, `control` the name of control file, `size` the number of members in the population, `pmut, pcross, sites, pinv` and `pressure` influence the genetic operators, `steps` is the number of rewriting steps for the L-System and `axiom` is the axiom for the L-System.

- **The number of members to be processed**: this parameter determines how many of the members of the population are processed in one iteration.

- **The name separator**: the names for matrix, production rule and string files are build from some components, for instance for the production rule file this is ‘‘`prodr`’’ + parameter three + “.” + member number, where the member numbers range from 000 to the second parameter (counting starts when an iteration is started). A name could for instance be `prodr000.000`, the production rules for the first member to be processed in any iteration (which iteration does not influence the name) where the third parameter was 000.

- **verbosity**: adding a fourth parameter (any) will put the program in verbose mode, some extra information is giving during running.

The command

```
calmgenalg_w simfiles/sim.tst 10 000 -v
```

would start the process using the parameter file `simfiles.sim.tst`, process 10 members each iteration, use code `000` in the names and turns the program to verbose mode.

During the process, the population of genetic codings of networks changes, ruled by the members with higher fitnesses. After each iteration information about, lowest fitness, average fitness and highest fitness of members in the population is written to file called `result.maprec`, which is stored in the directory indicated by the `file` parameter in the parameter file.

Before actually starting evolution on a population, one has to be created. This is done by the `calmnewpop` program. The command

```
calmnewpop simfiles/sim.tst
```

creates a new population with the size mentioned in the sim.tst file. The `calmnewalg_w` should be started on a population that was by the same parameter file as the one it receives itself.

7.6.2 `testxrand`

The `testxrand` program was written to give some graphic representations of both categorizations and network activations. The program is not as user friendly as could be possible, but a lack of time did not permit making a better looking, more user friendly version. The network should be coded in the program, which asks for at least a rudimentary understanding of the working of the CALM library and the program. Some features might be implemented as parameters for the program.

The command

```
testxrand sf <filename>
```

starts a simulation using one network with set parameters and evaluates this until stopped by the user (this can be done by simply killing it). At certain moments the learned categorization is saved as a picture to a file, exactly when, depends on when the user wants this to happen, by changing the program. This normally happens once every ten rounds of offering the patterns. This feature could be made a parameter in the future. The `<filename>` parameter is mandatory (leaving causes an error from the program). The file is written to directory `pic` (this should be a subdirectory of the directory the program is started from).

The command

```
testxrand ss
```

starts the same simulation, but the produced pictures are not written to a file, but immediately shown on screen. This does require an X-type terminal.

The command

```
testxrand v <filename>
```

shows a picture file written by this program. The file should be in the `pic` directory.

The commands

```
testxrand ?
```

and

```
testxrand h
```

print a help message.

Chapter 8

Parameters and Architecture

Apart from certain things as learning rules, type of connections and units, which are fixed within an artificial neural network paradigm, there are also variable factors, most importantly architecture and parameters [1]. To be able to solve a problem using an artificial neural network, we should first determine these two factors.

8.1 Architecture

In determining the architecture of the networks the first strategy used was the one used by Boers And Kuiper ([Boer 92]): determining the architecture of the networks by using Genetic Algorithms. However, this approach fell short. The parameters of the resulting CALM-networks were simply the standard parameters. These parameters appear to be unfit for networks that contain more than one internal module. It appears that not only the problem the network has to learn, but also the size of the network have an influence on which values for the parameters work well. At the moment the insight in what precise behaviour in networks is created by the changing of certain parameters is rather limited, which makes it hard to determine in a rational way what values the parameters should get to make the network work properly.

Therefore another method of determining architectures was employed, albeit a rather brute one: the architecture was developed by hand, keeping in mind what type of architecture might be useful for solving the problem at hand.

[1] We are talking about the normal situation here. There are cases in which we might want to change certain features of the paradigm, because we find that it is simply unable to properly solve the problem we offer, even though we want it to be solved.

8.2 Parameters

Almost all neural network paradigms incorporate some type of parameter, which influences the speed and the effectiveness with which they learn to solve problems. Most paradigms take only a few parameters, with the most important of those being the learning parameter, the parameter that determines how fast the values of the weights of the connections of the neural network can change.

CALM takes a rather special place when it comes to the number of parameters: the implementation used, uses no less than 37 parameters (of which a number, admittedly, are user defined). One of the reasons for the increase in the number of parameters compared with other paradigms is the fact that a CALM-network has more set weights; where most paradigms, like for instance Hopfield and Back Propagation Networks, have simple units (nodes), CALM has complex units: modules. Each of these modules has a set of set weights, for each of the eight different connections within the modules. These weights are normally set to the same value for all CALMs in the network, even though they can be chosen to differ per CALM. In fact, they should be chosen to differ per module when optimizing the parameter values of a network: preliminary research seems to indicate that fine tuning of the parameters should be done by changing parameters separately for each module.

All and all this leaves us with two rather complex tasks to complete before we can start using the network. Of course, we do have the possibility of keeping one of the factors, parameters or architecture, fixed, which is exactly what was done in a lot of experiments.

8.2.1 Determining Parameters

In trying to determine suitable parameters for a CALM-network, several approaches were chosen in this research.

Genetically

The research started out determining network architecture only using Genetic Algorithms, and choosing the parameters simply to be the default ones. However, it became clear quite soon that this approach sells the parameters short: it appears to be difficult, to say the least, to find a network that works well without in some way making changes to the values of the parameters: the network's performance is clearly dependent on the value of the parameters (or why would they have been implemented any way?). As suggested before, experiments showed that the default values for the parameters are fit for networks without internal modules, however, the networks developed using the genetic algorithm always contain internal modules, so some kind of alteration had to be made to the values of the parameters.

To come to terms with this problem we chose to try to determine the parameters of the network along with the architecture: codings for the parameters of the network were added to the chromosomes fed to the genetic algorithm, which were evolved along with the chromosomes for architecture.

This method, however, has a rather unfortunate drawback: for a chromosome to survive it needs to have a good architecture as well as good parameters. This results in good architecture with bad parameters and good parameter sets with bad architectures to be lost from the population. The parameters and/or architectures lost this way might of course be produced by the genetic algorithm again, but it will cost time. Since the evaluation of one a network can take a lot of time, due to the relative complexity of the CALM-paradigm, these type of experiments are not feasible on a normal Sparc or Indy machine. However, since a parameter set is only good for certain network architectures, the determination of architecture and parameters should best be done simultaneously

However, we needed a way to get around this time problem in an acceptable way. One way is to separate the stages of developing the network architecture and the parameters for the network. We first determine the architecture and later determine the parameters. For this to be feasible, though, we need some way to determine for which networks we want a good parameter set to be determined. Which sounds well enough, if it weren't for the fact that we have no safe way to determine which network architecture is promising. To be able to do some experiments with determining parameters apart from architecture a number of network structures were chosen (after being developed by hand), and for those the parameter space was searched using the methods described in the following subsections.

Manually

One approach was to change the parameters "manually". This sounds worse than it actually was. The first sixteen parameters were systematically upscaled or downscaled, separately: simulations were run where one of the parameters at a time was repeatedly multiplied by a factor of 1.02 for upscaling or 0.98 for downscaling. This was repeated 100 times, with each resulting network being trained five times, to filter out effects caused by random induced deviations. Apart from this, changing some of the parameters simultaneously for a couple of potentially interesting combinations was tried.

The most notable result from this experiment was that the parameters are quite robust: changing one parameter at a time appeared to have hardly any effect. Only two of the sixteen parameters resulted in serious affection of the network performance: as well downscaling the ninth parameter (the initial learning weight) as upscaling the eleventh parameter (the activation decay weight) resulted in the network not converging at all from a certain point on. Apart from this, changing some of the

other parameters seemed to have the effect of stabilizing the network performance: where tests with the original parameters tend to look erratic with large differences in performance between tests of one network, the changed parameters appeared to give smaller differences. The problem was, however, that this also resulted in worse performances: the performance was stable at a rather mediocre level (this is probably caused by the network not really learning, which would explain why all result are rather bad and constant). This, of course, is something we would not want: a couple of good solutions are what we want.

Evolutionary Strategies

In the experiments using evolutionary strategies a population of one hundred parameter sets was used. A new generation is created by dumping the fifty least fit individuals and make fifty new ones from each of the fifty most fit individuals. These were created by adding a random number, normally distributed round zero, to each of the parameters. To determine the standard deviation for each of these numbers, an extra set of parameters was kept, each of which was initialized to one eight of the standard value of the parameter [2]. The initial population was determined using twice the standard deviation, this to start out with a population distributed over a larger area. After each generation a random number was added to the standard deviation parameters with a mean of zero and a standard deviation of one fifth of the value of the parameter. Since the effect of a parameter with a negative value is different from one with a positive value, provisions were made for the parameters not to change sign (this was not necessary for the standard deviation, the absolute can simply be taken) The new individuals are evaluated by training the mapping problem (which will be treated in the next chapter) on the resulting networks. To avoid flooding of the population by one good member, the individual with the highest fitness was deleted from the population, although it did reproduce.

The advantage of using this type of genetic manipulation is that there is a closer relation between the objects manipulated and what they code: changing random bits, especially higher order bits, would throw a parameter far of from where it started, with this method the parameters evolve around their starting value, although a bad starting value would still be able to evaluate to a decent value (although it is severely hampered due to its bad start: the chance of the deviant individual disappearing from the population shortly after the start is huge).

A disadvantage is that it appears to be less biologically plausible: in real life gens information is coded in the abstract way information is coded in in bit strings.

[2] This resulted in the parameter sets being twice the size they were before, for each of the 30 parameters of the 100 members, a full set of standard deviations was kept.

Chapter 9

Experiments

9.1 The Problems

In this section some problems which were tested in this research or could have been tested during this research are visited.

9.1.1 The XOR Problem

The XOR problem is a relatively simple problem, which was mainly tried for testing purposes; it is so simple any paradigm should be able to solve it. The network gets two inputs, and returns true if exactly one of the inputs is one, zero else. As expected, CALM networks learned this problem perfectly. The used architecture can be seen in figure 9.9.

9.1.2 The Mapping Problem

The most important problem used during this research is a rather tough categorization problem, of which a simpler version was also used. It was originally designed by Van Hoogstraaten ([Hoog 91]) in order to investigate the influence of the structure of the network on its ability to map functions. He created a square $[0,1)^2$ input space. In this space 100 points (10x10) are assigned to one of four classes. The neural network has to learn with which of the four classes each of the 100 points is associated. How difficult it is for a neural network to learn such a problem is strongly dependent on the way in which the points are assigned to the four classes. If we have four 5x5 blocks for each of the categories, the problem space would be rather easy to separate. If we assign the points at random the network will have to learn each point separately. Figures 9.1 through 9.3 show examples of three mapping problems, the first with a clear division into the four categories, the second less organized, although still far from random, the third is very regular, although there are no adjoining points from

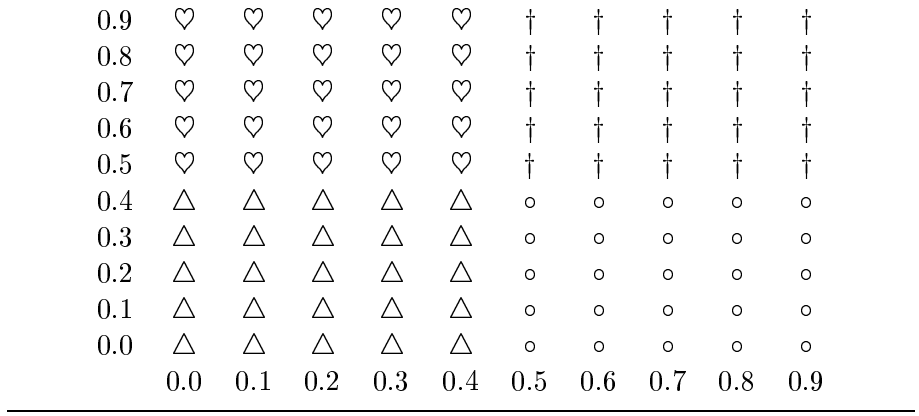


Figure 9.1: A mapping problem with a clear division into the four categories

the same category.

The first problem needs a cross type of separation between categories. From this observation and the fact that networks with unbiased input draw categorizations through the origin, we might conclude that what we need to solve this problem is input with bias. The second mapping problem has three points of one category surrounded by eight points of one other category; these points can be seen as miscategorizations. Originally (in Van Hoogstraten's research) these three points were miscategorizations, however, in Boers' and Kuiper's research as well as this research the network was supposed to learn the categories like this, in this driving the calculative power of the network closer to (or over) the edge.

9.1.3 The Spiral Problem

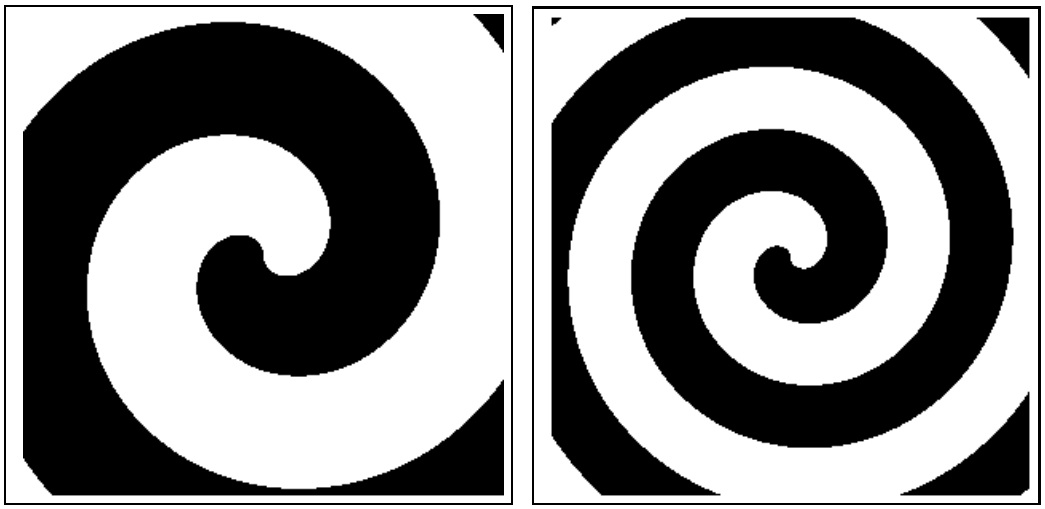
Imagine two flexible layers of different colours, for instance black and white. Put these on top of eachother and roll it up and take a slice of it. Looking at this slice one sees two entangled spirals. In the spiral problem these two spirals are used: take any point on this slice and determine on which of the two spirals this point is. Unlike the mapping problem, which is discrete, this problem uses a continuous space. This means that we have an unlimited number of possible inputs (abstracting from the fact that real numbers are represented in a discrete way on computers). In this research a function written by Bart Happel was used for calculating on which of the two spirals a certain point lays. The degree of spiraling (the tightness of the winding of the spirals) can be given as a parameter to this function. Figure 9.1.3 shows some examples of spirals. It should be noted that this problem is very difficult: the spiraling creates

0.9	∘	∘	∘	∘	∘	∘	†	♡	♡	♡
0.8	∘	∘	∘	∘	∘	∘	†	†	♡	♡
0.7	∘	∘	♡	△	△	△	†	†	†	♡
0.6	∘	∘	♡	△	△	△	†	∘	†	†
0.5	∘	∘	♡	♡	♡	♡	†	†	†	†
0.4	∘	∘	♡	♡	∘	♡	♡	♡	†	†
0.3	∘	∘	♡	♡	♡	♡	♡	♡	†	†
0.2	♡	♡	♡	♡	♡	△	△	♡	†	†
0.1	♡	♡	♡	♡	△	△	△	♡	†	†
0.0	♡	♡	♡	♡	△	♡	△	♡	†	†
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

Figure 9.2: The mapping problem used in both Boers' and Kuiper's research, the original developed by Van Hoogstraaten

0.9	∘	♡	†	△	∘	♡	†	△	∘	♡
0.8	†	△	∘	♡	†	△	†	♡	†	△
0.7	∘	♡	†	△	∘	♡	†	△	∘	♡
0.6	†	△	∘	♡	†	△	†	♡	†	△
0.5	∘	♡	†	△	∘	♡	†	△	∘	♡
0.4	†	△	∘	♡	†	△	†	♡	†	△
0.3	∘	♡	†	△	∘	♡	†	△	∘	♡
0.2	†	△	∘	♡	†	△	†	♡	†	△
0.1	∘	♡	†	△	∘	♡	†	△	∘	♡
0.0	†	△	∘	♡	†	△	†	♡	†	△
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

Figure 9.3: A very regular mapping problem, without large areas, in fact no two adjoining points are from the same category



(a) (b)

Figure 9.4: Spirals with spiralness 1.0 and 100.0, respectively. The origin is at the top in both spirals.

a total lack of linear separability, thus making it hard for artificial neural networks that are based on linear separability to learn to solve the problem.

9.2 GAs, L-Systems and CALM-networks

This research started out doing experiments using the same kind of development system Boers and Kuiper [Boer 92] used on their experiments. The initial intention was to let this project move along that way, however the initial results indicated that a different approach might be more fruitful. Because of the straight out disappointing results obtained in preliminary experiments using this set up, no definite tests were done (there was no hope for those to be good). This results in no real experimental results being available.

The object of solving any problem is not to simply find a network that turns the problem into a simple association problem, we want it to find the regularities in the input (which are definitely there). One way to aid this is to limit the number of CALMs in a network as well as the number of nodes in a CALM. The limits were set to 10 CALMs per network (apart from input and output) and 10 nodes per CALM.

The parameters used on the experiments using the GAs were (these are not parameters of the CALM-network, they are parameters of the genetic algorithm):

- population size 500, chromosome length of 1024
- rank based selection and replacement, pressure of 1.4
- mutation probability of 0.01
- crossover probability of 0.3, 10 crossover sites
- inversion probability 0.7
- L-System axiom A
- sites 3
- aging 1.00

Most parameters speak for themselves: the population size is the number of chromosomes in the population, chromosome length the number of bits in the chromosome, pressure indicates the willingness for selection to select a good solution, mutation indicates the chance for a chromosome to get mutated (flipping one bit), crossover indicates the chance that two chromosomes are crossed over, as inversion indicates the chance of a chromosome getting inverted. The axiom is the string from which the application of production rules stars, sites the number of cycles production rules are applied. Aging indicates an aging factor with which the fitness of each chromosome is multiplied each cycle of the GA (this should be 1 at most: higher values indicate that aging increases fitness, which is not something we would want). The aging factor was chosen to be 1, since initial tests did not indicate a positive effect of a different aging factor (0.99 was used). This last parameter was not implemented in Boers' and Kuiper's research.

These experiments were all run using the first two mapping problems

9.3 Categorizing Random Input

A good way to try to understand the working and the strength of a certain ANN paradigm is by offering it random input and see how it reacts to that. This is one approach that has been taken in this research: a simple CALM-network was offered a two node pattern: two random numbers in the range [0.00 ... 1.00]. This is fed into a two respectively eight node CALM, which then categorizes to one of two, respectively one of eight categories. On learning the network was offered 100 pairs of input. The categorization of the input can then be plotted by simply offering points on a grid, in this case a 200x200 grid for the two category examples and a 400x400 grid for the eight category examples, and plot dots with unique colours for all categories. The images that are created like this give a good idea of the potential complexity

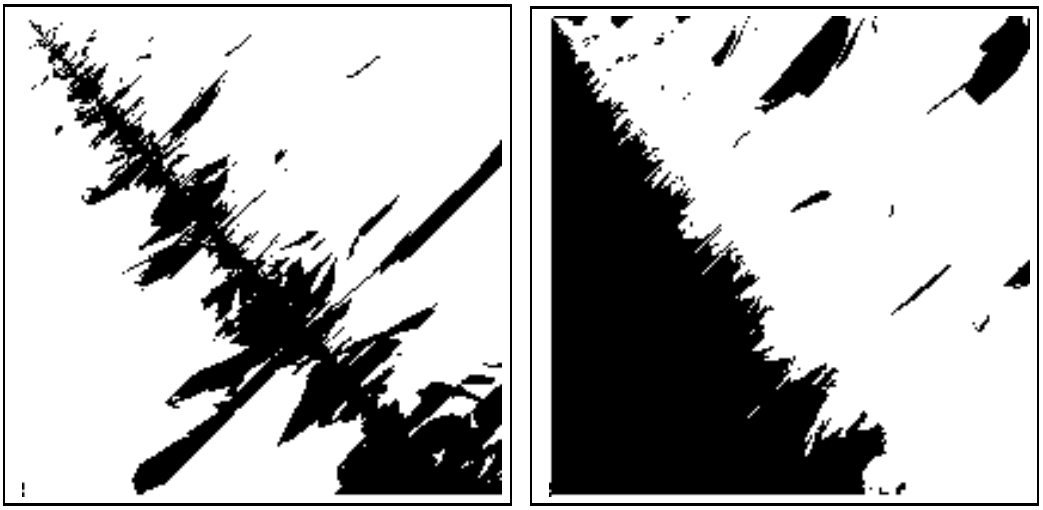
of categorizations CALM-networks can achieve, as can be seen in figures 9.5 through 9.7. In learning new random input was used for each time a pattern was offered. Figure 9.8 shows a categorization of the mapping problem that gives a categorization that looks to be heading to a good categorization. Unfortunately, the network did not improve towards a perfect categorization from this point. All tests were run using the network from figure 9.9, and using the standard parameters.

The irregularity of some of the categorizations suggests that a CALM-network should be able to learn virtually every problem: there appear to be almost no restrictions on the complexity of the categorizations. We should, however, get some kind of control over these categorizations, or the potential strength is lost.

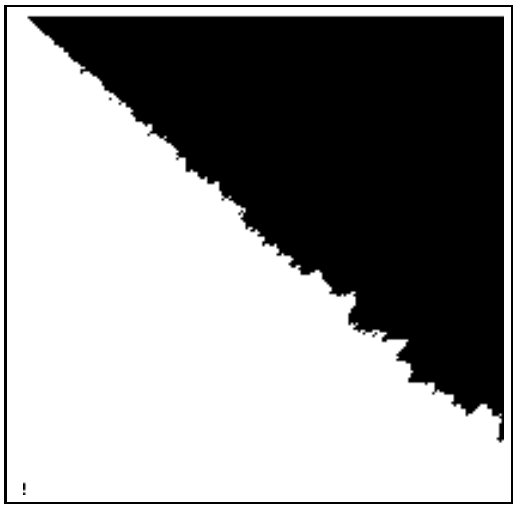
9.4 Visualizing learning

One of the problems of the complexity of CALM-networks is that it is quite hard to get any conception of what happens inside a CALM-network. To come to terms with this problem some graphical method has to be used (the repeated activations present such a vast amount of data, that leafing through is hardly a manageable –or efficient, for that matter– method). The method used for this was a rather simple one: each node of a module is represented by a circle, with a disc inside representing the activation of the node in question. A CALM-module is represented by two horizontal lines of nodes, the R- and V-nodes, adjoined by two loose nodes, the E- and A-nodes. The lowest line of modules represents the input patterns (a pattern is actually a stripped version of a CALM: all nodes but the R-nodes are omitted), the middle the internal modules (all of them, not minding the connections between modules) and the top line represents the output module, as can be seen in figure 9.10. The connections between the modules were left out because of the potential complexity of these connections, as well as the overhead created by positioning the modules in a reasonable way. Figure 9.9 shows the architecture of the used network. Figure 9.10 shows part of the categorization of one input pattern. A larger example can be found in Appendix B.

From these graphical plots it becomes clear that the activation caused by the input pattern or patterns is propagated through the network, in fact resembling the way this happens in BPNs rather closely. However, instead of there being loose nodes receiving activation, as in BPNs, the R-nodes of a CALM-module receive activation, which they propagate through other modules, as well as categorizing it themselves, after which this propagation is passed on to the other modules as well, of course. This modularization gives the network the opportunity of splitting the problem to be learned in smaller parts (albeit that networks find it hard to do this by themselves, which, unfortunately, is not an uncommon problem).

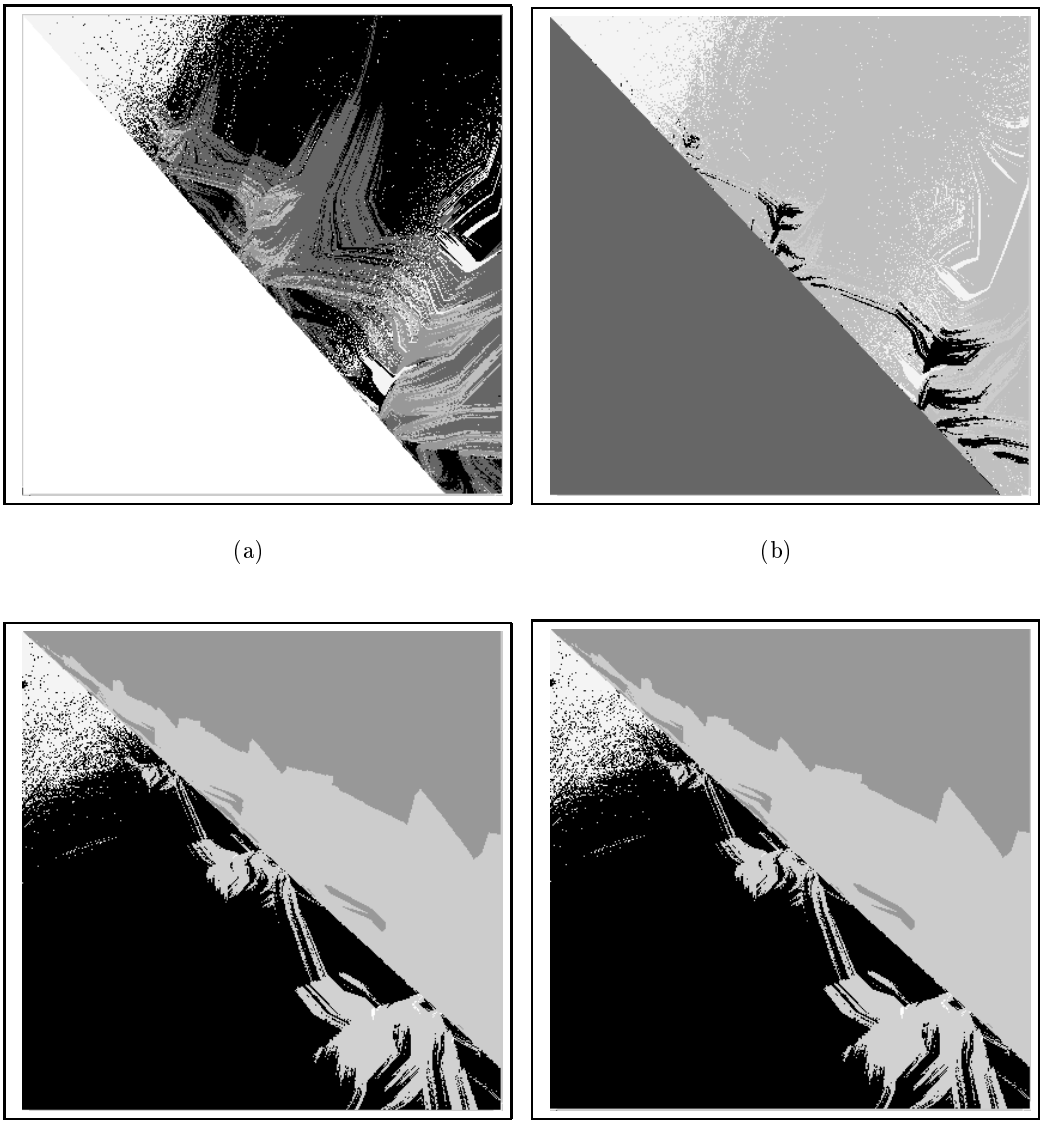


(a) (b)



(c)

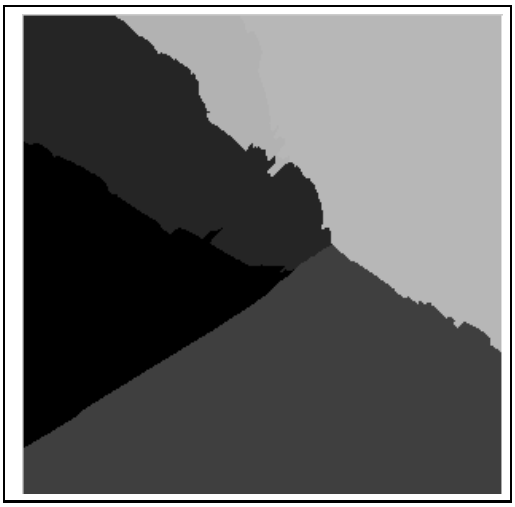
Figure 9.5: Some rather striking examples of categorization of random input to two input categories. (a) gives a good view of how complicated the borders between category bounds can be, (b) shows a categorization closer to the "ideal", two categories separated by the diagonal whilst (c) shows a near perfect categorization. After more learning cycles the ragged borders tend to transform themselves into one like the one in (c). All images are created by networks without a bias node, which can be concluded from the hyperplanes going through the origin (upper left corner). The small black spots in the lower left corner were caused by a somewhat inaccurate snap, not by the network.



(a) (b)

(c) (d)

Figure 9.6: Letting CALM-networks categorize input to more than two outputs can result in stunning fractalic images. Examples shown have categorized to eight categories, represented by eight gray scales. (a) through (c) show categorizations after one learning cycle, (d) after two cycles, same run as (c), that is, (d) is the output from the same network as (c), but after more learning. Since (c) was already close to what seems to be a "normal form", it did not change very much. All images are created by networks without a bias node.

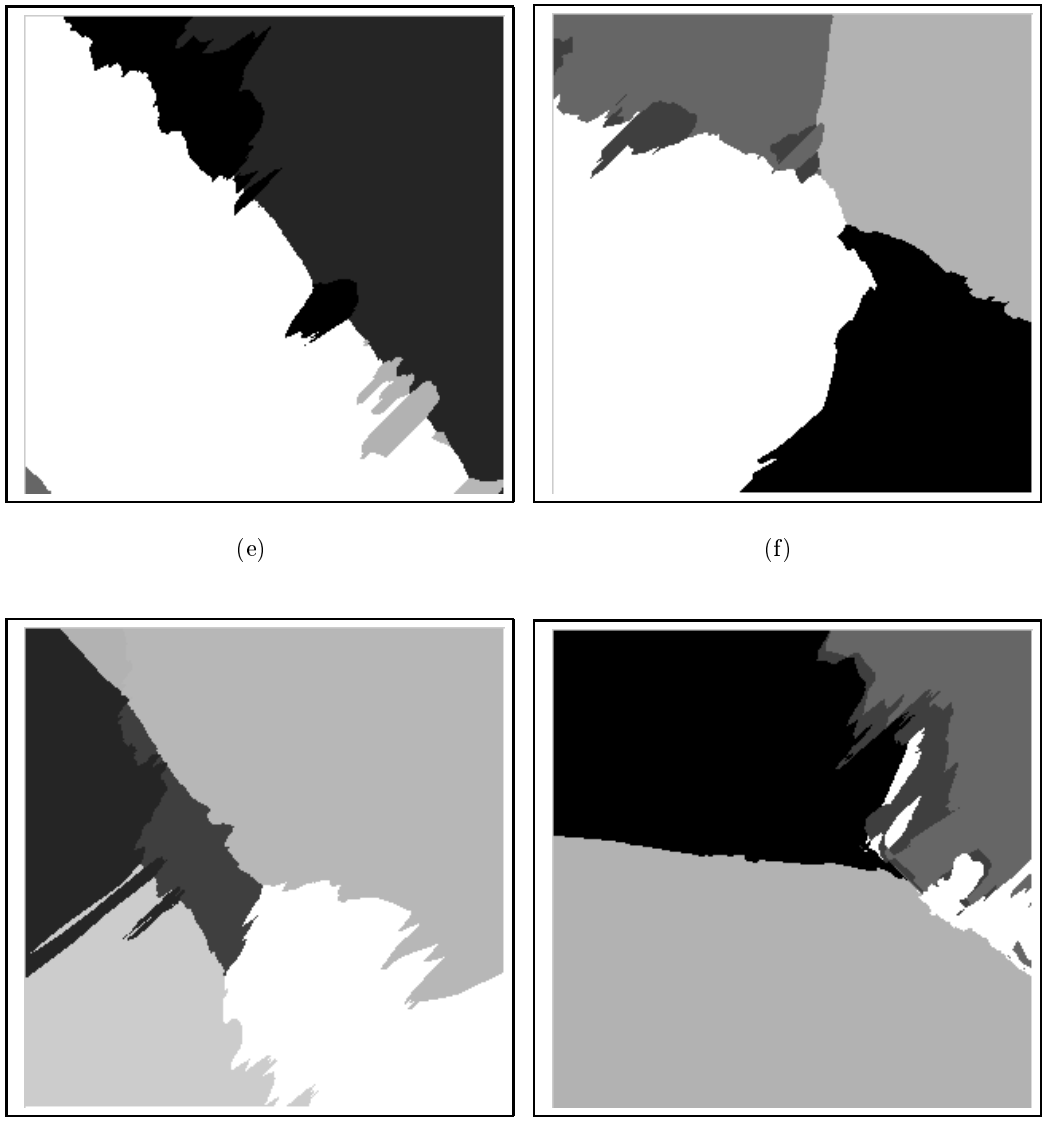


(a)

(b)

(c)

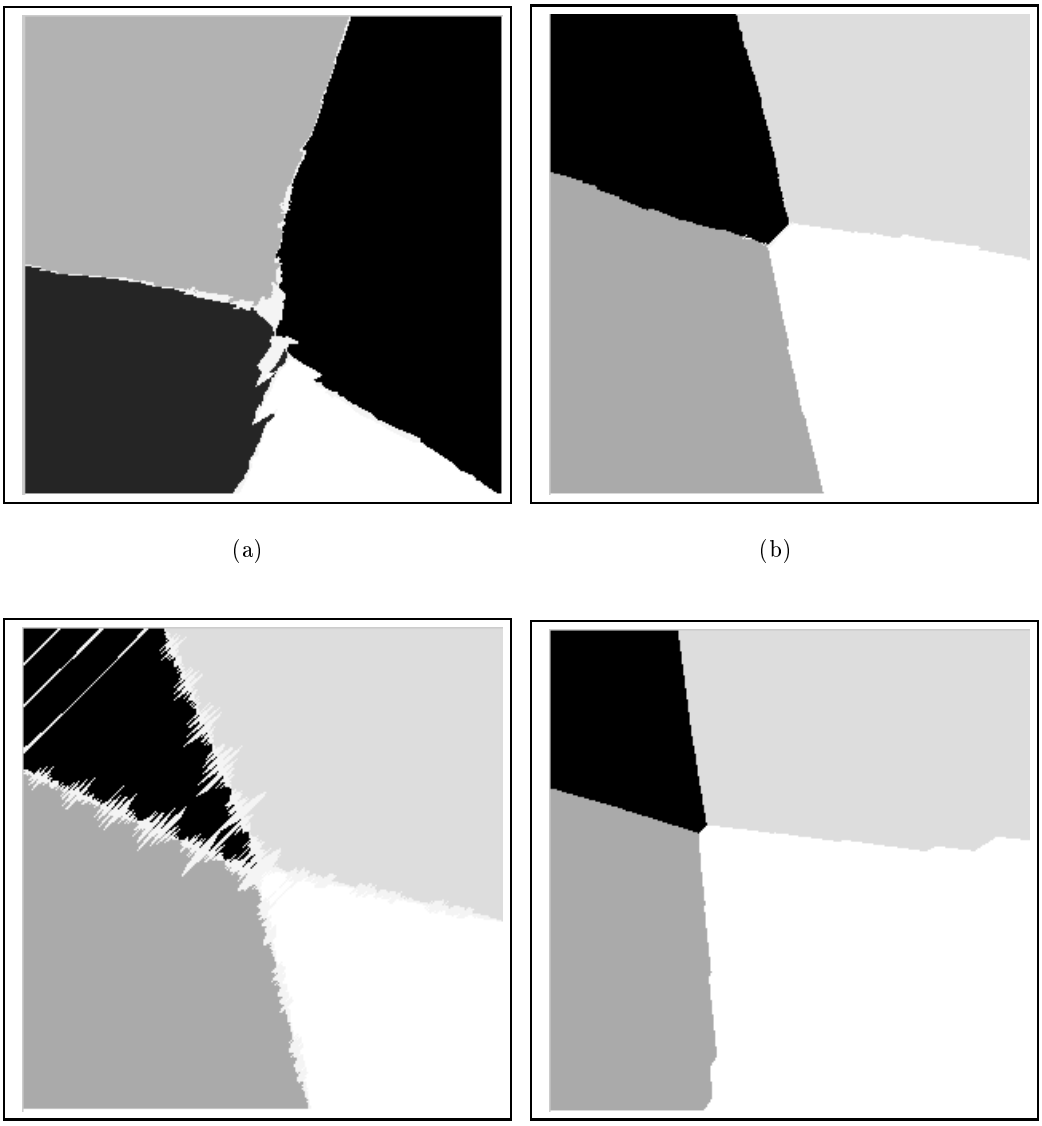
(d)



(e) (f)

(g) (h)

Figure 9.7: Some examples of random categorizations to eight categories, using biased input. As can be clearly seen, these categorizations are far less centered round the origin than unbiased input categorizations are. All these categorizations were made on the same run, in this order.



(a) (b)

(c) (d)

Figure 9.8: This figure shows categorizations of the simplified mapping problem. Although these categorizations are far from perfect, the tendencies to grow towards a quadrant categorization can be clearly observed. (b), (c) and (d) were taken from one run, in that order, which seemed to be doing well right away. Unfortunately it moved away from the perfect categorization. This could be caused by overcategorization, on the other hand, the network never learned to solve the problem perfectly, so more training was necessary (and did not have the desired effect). The architecture used can be seen in figure 9.9

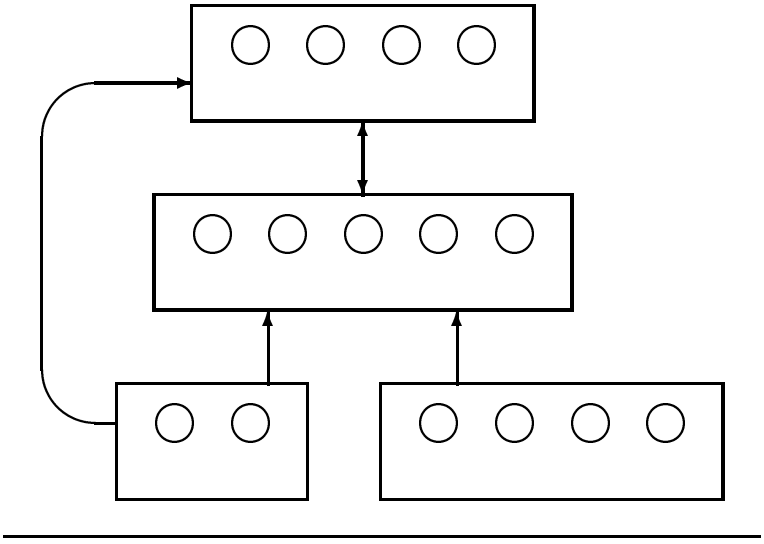
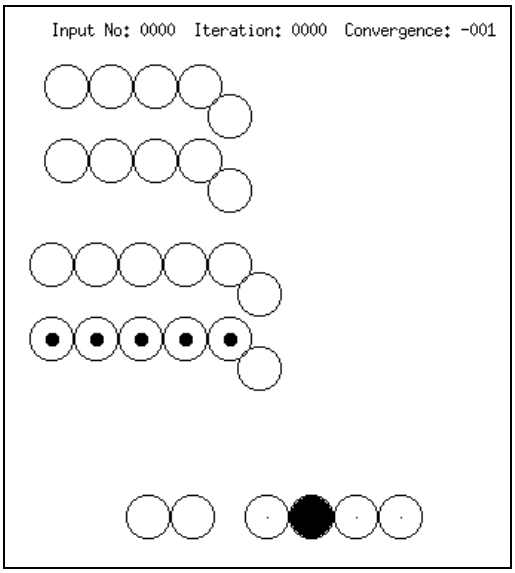


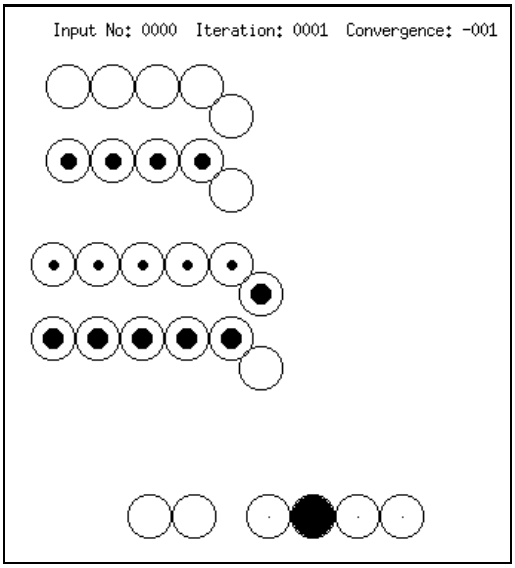
Figure 9.9: This figure shows the architecture of the network used in figure 9.10

9.4.1 Mapping Problem Tests with Evolutionary Strategies

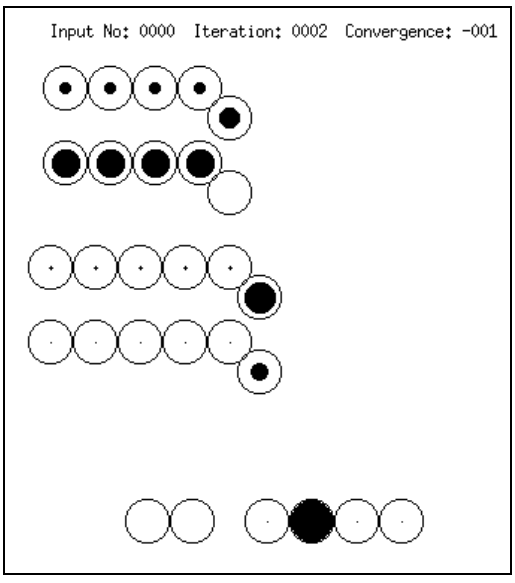
For the more elaborate tests in this research, the following set up was used: the networks used were one of those in figure 9.11. A pool of fifty parameter sets was generated, each parameter with its own standard deviation, of which all parameters of all members were adapted by using evolutionary strategies each generation. To each parameter and each standard deviations one of these numbers was added. The numbers had mean zero and initial standard deviation one eigth of the value of the parameter. Apart from this, a scheme was tried with uniformly distributed parameters in the $[0.95 \ldots 1.05]$ range, which were used as factor for the parameter values. This approach was first taken because of initial implementation problems with the normally distributed numbers set up. The fifty new members were evaluated, after which the worst fifty of the one hundred individuals were discarded. Each of these members was evaluated by counting the number of correct categorizations on the simple mapping problem. Although the normally distributed set up would give for a better range


Input No: 0000 Iteration: 0000 Convergence: -001

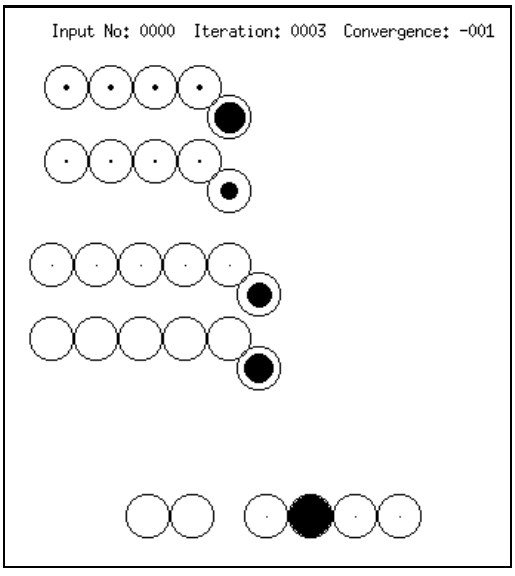
(a)


Input No: 0000 Iteration: 0001 Convergence: -001

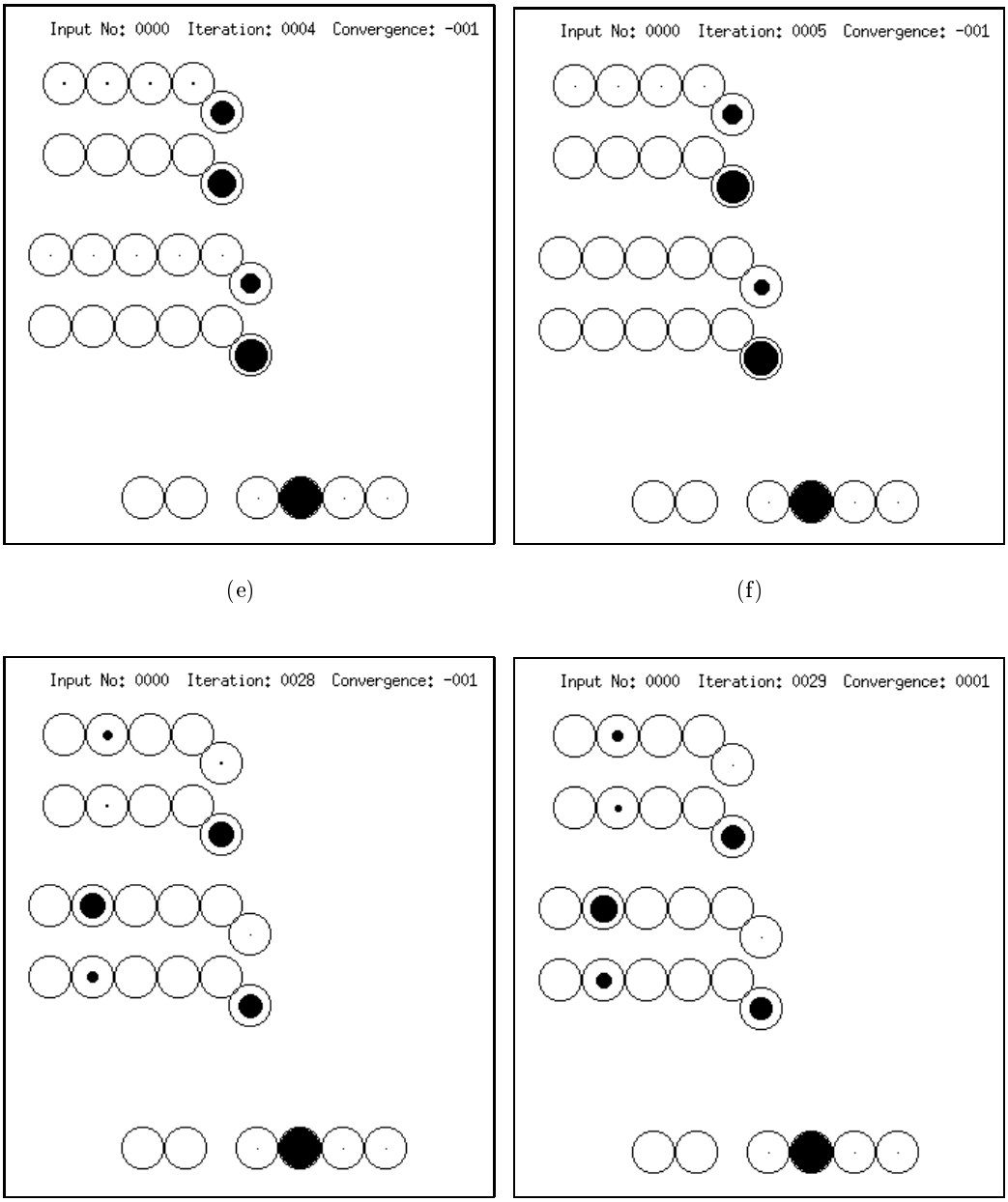
(b)


Input No: 0000 Iteration: 0002 Convergence: -001

(c)


Input No: 0000 Iteration: 0003 Convergence: -001

(d)



(e) (f)

(g) (h)

Figure 9.10: This figure shows the more important parts of the categorization in a CALM. (a) through (f) show the initial surge, while (g) and (h) show the activations shortly before categorization. See Appendix B for a more elaborate treatise of the total process. The network structure is given in figure 9.9

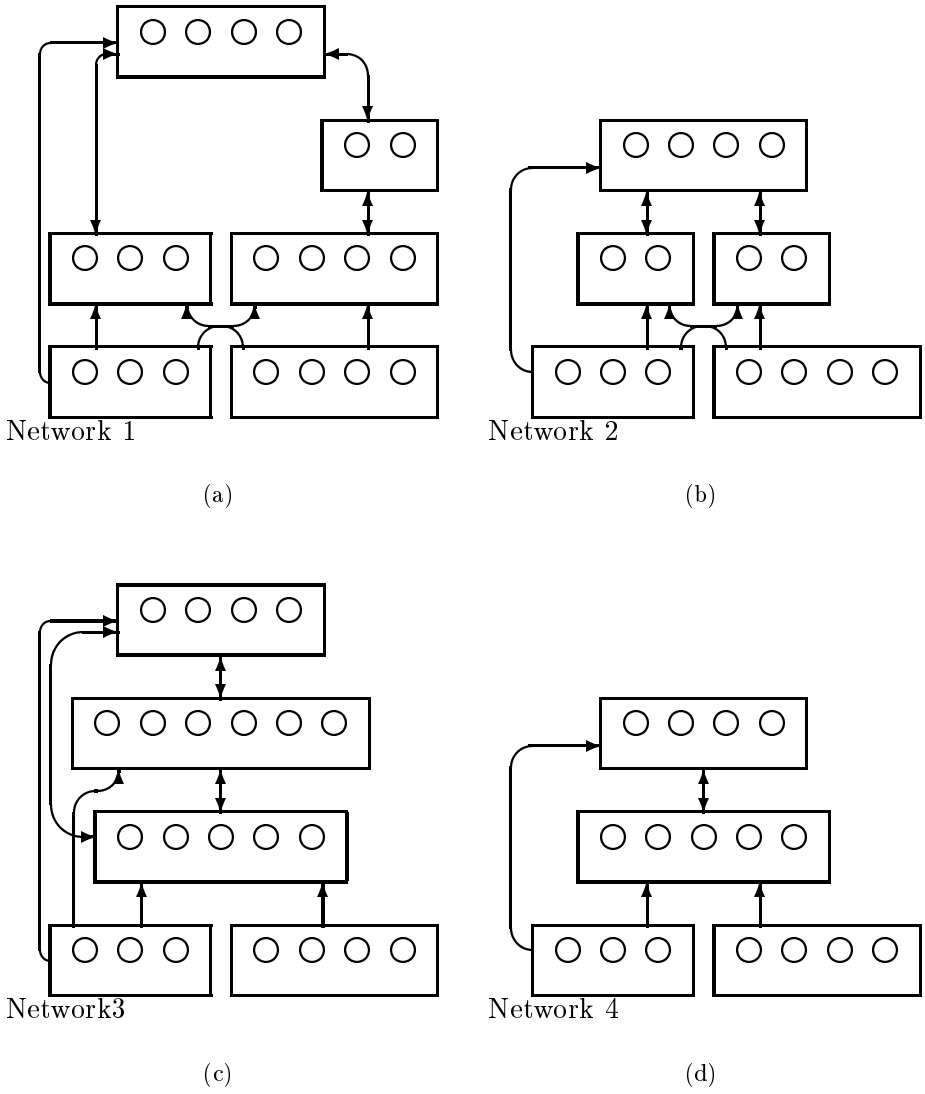


Figure 9.11: The most important handdesigned networks used in this research for the mapping problem.

of parameter values, there are no really noticeable differences between the set ups. Apparently the spreading of the parameters itself is most important.

Figure 9.12 show the main results of the tests run. It also refers, along with the networks in figure 9.11, to network 5 through 8. These networks have the same architecture as figure 9.11(d), except for the number of nodes in the internal CALM: network 5 has three internal nodes, network 6 has twenty, network 7 has ten and network 8 has four.

		config			
network	bias	initial parameters		changing parameters	
		pre-activation	extra input	pre-activation	extra input
1	y	44	50	80 (25, 27)	50
2	y	58	0/48[1]	79 (27, 174)	82 (2, 30)
3	y	45	40	63 (13, 13)	90 (53, 57)
4	y	48	0/48[1]	72 (14, 26)	82 (1, 4)
5	n	57	0/43[1]	74 (10, 12)	90 (191, 284)
6	n	29	0[2]	82 (6, 10)	87 (16, 26)
7	n	25	0/46[1]	78 (7, 122)	86 (22,210)
8	n	62	0/48[1]	89 (50, 231)	88 (135,256)

Figure 9.12: The most important results in the evolutionary strategy tests. The numbers are the number of correct categorizations out of 100, the numbers between parentheses are the number of generations after which the best member was found, followed by the total number of generations for which the simulation ran.
[1] These networks did not reach an unambiguous categorization, therefore their fitness is zero, the number behind the slash indicates the number of inputs categorized correctly, for those instances that were categorized. It should be noted, though, that these numbers are highly inaccurate and are only given for the sake of completeness.
[2] The networks did not converge with default parameters.

When the method of learning by using a second input pattern representing the desired output was used, all networks that didn't achieve an unambiguous categorization (that is, network that failed to map at least one category to an output node, or had mapped more than one category to a node) were discarded. It should be noted, though, that the requirement that every category is mapped to a unique node, does not necessarily mean that all the categories are used in the division into categories of the problems. During several simulations done, networks that had mapped all categories to unique nodes turned out to not use all the categories for the categorization

of the input. These networks were accepted. It should be noted that this type of mis-categorization is impossible in the pre-activation tests: only the pre-activated node is accepted as the correct answer.

The large number of members in the populations made it possible for the simulation to find the best member after a rather short period of time. Despite this occasional, and quite marginal –one or two points better– score improvements were found at times. Compared to some of the scores Boers and Kuiper found ([Boer 92]), these scores, although rather good at times, are quite disappointing.

Especially network 2 has an architecture which one would expect to be very suited for the problem at hand: dividing the input into four quarters. One module could separate left from right, the other top from bottom. However, the score this network achieved, fails to show this enhanced "fitness". In fact, the only way a CALM-network seems to be able to learn to solve a problem with such rigid borders and lack of fuzziness between categories, seems to be learning each of the offered patterns separately. This would be a reasonable explanation for the fact that the networks scored more or less the same on mapping problems 1 and 2, even though the first one is a lot easier than the second one: the first is an easily generalizable problem, while the second can only be solved with some sense for detail (especially the seemingly misabled points in it). This is further backed by the fact that network 4 scored 87 correct categorizations without using bias on the spiral problem, which should be markedly more difficult than both mapping problems.

The fact that there is hardly any difference in performance between networks using bias-nodes and networks not using bias nodes can probably be found in the fact that performance is rather poor anyhow. Bias-nodes are direly needed for precise and complete categorizations, however, if a network does not categorize the full set correctly anyway, the presence of a bias-node is of less importance.

One big difference between the pre-activation method and extra input pattern method can be found in the reaching of maxima. Where the extra input pattern method almost all of the time found several (if not a lot of) members that were on or close to the maximum, maxima in pre-activation were often clearly above the rest of the field. In several cases the difference in correct categorizations between the fittest and the second fittest member was more than 10. Apparently, the values of the parameters with which the network performs well are more strictly defined with pre-activation than with the extra input pattern. This might also explain why the difference in maximum over the different networks is far larger than it is in the extra input pattern tests. The fact that high scores are more irregular in pre-activation makes it less predictable, and therefore raises the chance of finding a member with a score close to 100. However, the highest scores so far are below those found with the extra input scheme, so that the irregularity might only result in less good solutions being found, without them having a higher score. Apart from that, the pre-activation networks find good scores quite seldom, which also becomes apparent from the fact

Figure 9.13: A typical division of the problem space for network 1, using fixed standard parameters.

that the simulation that ran real long still achieved their best score relatively early in the simulation. The poor performance of the extra input networks with initial parameters, seems to indicate that these values for the parameters, as suggested before, are not fit for networks with more modules.

There is no way of predicting when a good solution will be found for the first time (that is, a solution that comes close to the best solution found).

9.4.2 Connections

One result that is clear from experiments is the best connecting pattern from the input patterns to the output module. A network works best when only the real input is directly connected to it.

The reason for this is, of course, quite obvious: if the supervization input is directly connected to the output module, this doesn't need pay any attention whatsoever to the internal modules, the direct connection with the supervised input suffices. This, however, does not necessarily mean that this happens in practice: networks sometimes come up with better scores when both supervization and real input are connected then when supervization input only is connected. It therefore is not possible to draw any clear conclusions on which method works better.

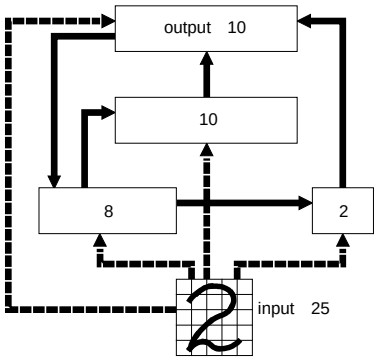


Figure 9.14: One of the networks used in Happel's research

9.5 Handwritten Digit Recognition

Over the past couple of years the psychologist Bart Happel has been working on a dissertation on CALM (see for instance [Happ 92]). One of the problems he used during his research was a handwritten digit recognition problem. This type of problem is very suitable for a categorization type network like CALM: there are ten categories (one for each digit), in which all scribblings have to be put.

In solving this problem he used a network with an architecture found by a GA and genetically manipulated the learning parameters of this network. To be able to test this network with he attached an input unit where the user can write a digit on screen by using the mouse. This input is sampled on a 5x5 square: the more "ink" in a square, the higher the value. Each of these squares is one of the 25 input signals to the network. On testing half of the input is used for training, the other half for testing. The network turned out to be quite robust in recognizing: a digit had to be seriously mutilated for the network not to recognize it anymore. Figure 9.14 shows a preliminary network used in Happel's research. This network reached 81% correct responses on the 100 digits large test body, with that falling seriously short of the BPN found by Boers' and Kuiper's method: they found a BPN without a hidden layer that found 97% correct responses. The CALM network however does not suffer from the BPN problem that learning more problems seriously interferes with previously learned problems.

This problem wasn't tested during this research, although it should be noted that it is an interesting experiment to run in the future, since it can not only be compared to Boers' and Kuiper's genetically architectured BPNs, but also with the CALM-network Happel used.

Chapter 10

Conclusion and Recommendations

As argued before using genetic methods to determine the architecture of a network relieves us of the problem of having to design a suitable architecture for a problem by hand. The same, although less strongly, can be said for parameters: even though we do have more insight in the function of the parameters than we have in the function of architecture, fixing them to optimize the learning of the network is still difficult. The beauty of the used scheme is that all we have to do, essentially, is offer the problem in the right format and the system will look for the best network itself (using the criteria we give it).

The other side of mentioned independence of the system is that it becomes even harder for us to determine what's going on: we could not work that out when we were just using a hand designed artificial neural network, but now that we do not even develop the network for ourselves, we can only look at what comes out. This would not be so bad, if it weren't for the fact that a lot of people do not wish to accept results given by a system they are unable to understand, and with the system getting more complicated, acceptance will be harder to obtain. All we can do for now is hope that further research will make clear what happens inside an artificial neural network.

This said, we have to look at the results of this research, and can not go around concluding that those are rather disappointing. Even though some of the problems offered to the networks are rather simple, the CALM-networks did not perform particularly well on them.

A first explanation for this could be the fact that determining a suitable architecture as well as a set of parameters to solve a certain problem, is by no means easy, in fact, it has been one of the most difficult problems within biologically based methods. The fact that the parameter set of CALMs is so much more complicated than it is for most other Artificial Neural Network paradigms, makes this particular case an

even more difficult one. Mainly because of this, networks developed by hand were used, thus alleviating the genetic algorithms task, since it has only the values of the parameters left to determine. This did lead to better results; most often scores of 80-90 out of 100 were obtained. Pretty good scores, but still not living upto par.

10.1 Conclusions

The idea behind the research for this thesis was to obtain a more efficient way of coding both architecture and parameters for a CALM-type artificial neural network. In order to do so genetic algorithms (GAs) were used. However, if GAs were used sec, the chromosomes representing networks would have to grow with the growth of the network. To circumvent problems the networks were not coded themselves, but codings of an L-Systems type grammar, with each L-system representing the structure of a network, were used. As a result fixed size chromosomes could we used, if larger networks were needed all that was to be done was increase the number of steps of the L-system, thus giving it the chance to lengthen the string representing the network.

One thing that remains to be solved for sure is the amount of computing power needed. Finding networks even for simple problems takes a lot of time, simply because of the fact that to judge the quality of a network the network needs to be tested. However, this great use of computing power can be seen as an investment: once a good network has been found, a lot of time can be saved. If a network has to be used often, the amount of computing power needed for the network itself is also of great power, the further this can be limited, the greater the profit on the investment.

From Bart Happel's work it follows that CALM-networks do quite well on certain problems: his digit recognition experiment got good results on both recognition and generalization of input. Yet a network of the same size (for internal modules, that is. The size of input and output modules is determined by the problem the network is to solve) is unable to come to a reasonable score on even the simple mapping problem. A possible explanation for this can be found in the kind of problems: digit recognition has a very large problem space (the input is 25 dimensional), and a lot of room for fuzzy borders with it: the room between members of each of the categories is so large that CALM is provided with ample room for error. On the mapping problem, on the other hand, we have a limited problem space (2 dimensional). Moreover, we expect the networks to produce a very sharp distinction between the inputs. Unlike with digit recognition, the mapping problem hardly has undefined areas (the points in the problem are quite close together). Apparently CALM-networks are unable to produce a division of the problem space in such a way that a well performing network is found.

We must remember why we started developing ANNs, though. From the psychological point of view, if was done to obtain better insight in cerebral activities. From the computer science point of view it was done to develop tools for solving problems that are hard to solve with normal algorithmic programs. People are generally to

very good at solving the type of problems used in this problem, that is, we would not perform very well on the difficult mapping problem (although we would on the easy one, apparently what what takes us ahead of CALM is a better capability of generalizing: we would simply remember the four quadrants, while the CALM-network seemingly attempts to remember the individual cases) or the spiral. We have high expectations on CALMs performance on these type of tasks, because we know BPNs solve them well enough, but it really is more important that ANNs perform better on the type of problem computers find hard to solve. The scores obtained by Happel on digit recoginition (90% succesfull) are promising in that line, although more research will be necessary.

The overall conclusion must be that CALM-networks, as yet, are unable to properly solve certain types of problems. Large problems with vague problem areas are solved well enough, yet the scores on small problems with tightly defined solutions are not good enough to compete with, for instance, Back Propagation Networks.

10.2 Comparison of CALM and BPN

When we want to compare two paradigms, the first thing we need is one or more criteria to judge the quality. In this section three criteria will be attended to.

The most obvious criterion is the number of correct categorizations the network gives. This criterion is easy enough, however, if a very small network of one type gets 97% correct while a giant of the other type gets 99% correct, one might wonder if this isn't too simple a criterion. As mentioned before we want to have networks that do not simply learn an association, but networks that actually learn what features in the input are of importance for solving the problem. For this we have another criterion.

Determining which of two networks, a CALM-network and a BPN is biggest, we can't simply count something, since the paradigms do not use the same kind of units. We could count the number of modules: the CALMs in a CALM and the modules at the lowest level in BPNs (as explained in an earlier chapter). A measure we could use is the number of nodes in the hidden layers of the BPN, compared to the number of R-nodes in the internal CALMs. Since input and output width are equal (at least we hope so), these are of no importance.

A final criterion in comparison is the number of iterations each of the networks uses. As with the number of nodes, however, the correspondence is not a direct one. Where a CALM normally converges (reaches a categorization) within some 30 steps, a BPN uses just one. However, it may take several thousand offerings of the problem to a BPN before this has learned the problem, where a CALM might have learned it after just one offering. What we could compare is the number of occasions on which a pattern was offered to a BPN with the number of occasions on which a pattern was offered to the CALM-network multiplied by the number of steps it took the network to converge to a solution.

Another, and much more simple criterion, would be to simply take the amount of computer time the respective networks needed to learn the problem.

Having said all this, we come to the rather sad conclusion that all these projections may be superfluous: on the problems tested in this research, the results obtained on runs with CALM-networks are far less promising than those obtained by Boers and Kuiper ([Boer 92]) using Back Propagation Networks. They, for instance, found networks with a 100% score on the difficult mapping problem, while the maximum obtained in this research was a somewhat disappointing 90%.

10.3 Recommendations For Further Research

The research report to which this research is a sequel ([Boer 92]) already mentioned a number of topics on which further research might be useful. However, since this network started out by using a different paradigm and time was short, these recommendations did not find a place in this research. This results on some of the recommendations in genetic algorithms being identical to those mentioned in [Boer 92].

10.4 The genetic algorithm

Make a more stable fitness function. The fitness function currently used is dependent on training the network once. However, due to the random process at work in learning, the same network would not on every training section obtain the same fitness. This means that a certain structure can gain a far too important place in the population due to a once-in-a-lifetime-type push in the right direction by the random processes. To prevent this from happening the most fit creature in the population might be evaluated each step, averaging its fitness over all evaluations. This, however, would take more computing power still, when so much is used already.

Find a better way to genetically manipulate the parameters. At the moment the learning parameters of the network are manipulate by the genetic algorithm in a way very similar to the way the chromosomes for the architecture are manipulated. It might be better, though, to manipulate those in a different way.

Evaluate a network once. At the moment when a network is found it is evaluated right away (excepting the axiom). This, of course, is not too efficient: it would be far better if networks that have been evaluated once, are not evaluated again, or if reevaluation would be used to make the fitness a more reliable one. To achieve this, a database would have to be kept of the networks already evaluated. However, there might be a problem in determining which networks are the same: if two strings form the same network, they are not necessarily the same. If two networks are the same in structure, their matrices will not necessarily be the same: nodes or modules might be interchanged, thus change the matrix without changing the actual

structure. At the moment it is hard to judge were implementing this would save more time than the overhead required uses.

Calculate the fitness of the initial population. The initial population is created with random chromosomes. The fitnesses of all these chromosomes were set to zero. However, it might be that these initial members of the population have certain qualities that enhance the fitness of the population. These qualities would be lost, since the chromosomes with fitness zero are gradually pushed from the population (at least we hope so). Again, this is something of which it is unclear whether the extra amount of work required is justified by the information won from it.

Initialize the population with members having a positive fitness. Instead of using an initial population with random numbers, of which a lot turn out to have a zero fitness (this could be observed from the new members made during genetic evolution), we might use initial members that have a positive fitness. A large part of these zero fitness members create an L-System that has no rules that can be applicated. A better way to solve this problem would be to change the axiom: the larger the axiom, the greater the chance a production rule can be applied. This, however, would seem like tampering with genetics. A better way to solve this problem would be to evaluate a string before inserting it in the initial population and only place it in the population if it has a positive fitness. This might also be a way to get a more diverse population, since the initial population contains far more members that can have a positive effect on the evolution.

Use a set of useful production rules in the initial population. An extension of the previous idea would be to make a set of useful production rules, code this set as a chromosome and insert this in the population. This method poses a threat, however: since we do not know for sure which structure will emerge as the fittest member, tampering the population this way might steer it in a direction away from good solutions, thus making it much harder for the GA to find a good solution.

Include the axiom in the genetic process. At the moment a fixed axiom is used. Including this axiom as a separate gene in the population, might create new axioms from which very useful solutions, that are otherwise neglected, can start off. This would also be more biologically plausible: the human ovum can be seen as a type of axiom which contains the information on which the coded production rules can operate.

Find a tighter relation between GA-string and network architecture. "Real life" chromosome "bits" code a feature of the organism directly, that is, there is a direct relation between the possible development of a certain feature and such a "bit" (the bit determines the genetic coding of the feature, the genotype, from which the organism, the phenotype, develops). Artificial chromosomes are represented by a string of characters, with each character having its influence on the architecture of the network which develops from it. However, the phenotype in natural organisms is similar to the sequence of characters coded by the chromosome, and not so much to

the architecture of the network. Making the relation between the chromosomes and the architecture of the network more direct might cope for the networks to be more flexible, thus giving them a greater potential power.

Apart from that, the use of Genetic Algorithms might work better if the correspondence between chromosomes and architecture was closer, than it did in this research, since the problem presented to the Genetic Algorithm appears to be too large to be taken on succesfully.

Use larger computers. This may sound like the another case of software crisis, however, the problem at hand simply asks for a lot of CPU time, the faster the CPU(s) used, the sooner the problem is solved. One of the reasons no good networks were found in this research might have been that the simulations weren't run for a long enough period of time.

10.4.1 CALM

Use incremental learning. So far the networks were offered the complete problem immediately. It may, however, work better to offer bits of the problem at first. CALM-networks do solve the XOR-problem, which has a problem space similar to that of the mapping problem. The networks might work better if the network first gets offered the corners, gradually working towards the center of the problem space, thus guiding the network along the way.

Something similar holds for the spiral problem. The problem in itself is very hard to learn, yet it might learn better when the spiral is offered bits by bits, starting of with a filled circle in the middle, and only the outside of the category. The second spiral can then insert itself into the first gradually.

For this approach to work well, the CALM-network must have some sort of sense of what the problem is, it should be more than just a collection of inputs, as the mapping problem sometimes appeared to be treated. If the network treats the problem simply as a collection of patterns, this gradual learning scheme will not help, since all it will see is changes in points (or at least we hope it will), without noticing the connection with the rest of the inputs.

Enhancing strict learning. The pictures of random input shown in figures 9.5 through 9.7 suggest that CALMs are able to make sharp distinctions between categories (although some of the borders suggest a certain degree of fuzziness: the networks with more output nodes show a kind of "smudges", these might be an indication of doubt about the category of those particular points). This suggest that CALMs might be able to solve the type of problems offered in this research, but that the learning method is not sufficient to reach through to the network. Despite this, some rather wild categorizations were found on non-random problems, but unfortunately those categorizations didn't come close to the desired categorizations.

Further evidence for a lack in strict learning comes from the fact that a lot of the

categorizations produced by networks were definitely attempts at making a quadrant-type categorization, but lacking the exact boundaries.

Do more theoretical research. Due to the complexity of CALMs, still relatively little is known about the internal processes. It remains largely unclear what exactly the effects are of certain parameter values and which values enhance learning in certain architectures or with certain problems. This lack of knowledge about the theoretical background of CALM makes it very hard to make clear judgements about results. In fact, this research suffered a lot from that: the networks were expected to perform well, yet they did not. The first thought in that case is that there is something wrong with the programming, while in the end it turned out to be the paradigm that was not working well enough.

Try different connecting methods. All connections in CALM-networks are full connections. Even though the learning algorithm can adapt this for connections between CALMs, by making a weight zero, this does make learning a giant proces. Apart from that, all weights within a CALM are fixed during learning. Using different connections (for instance one-to-one, as mentioned before) or using less rigid internal weights might aid learning.

Check the convergence methods. Learning in a CALM-network depends on the convergence criteria for an important part. If a network is assumed to have converged while it hasn't, chances are the result will not be very good. Some of the categorizations, see for instance some of the pictures of categorizations of random input, are very ragged, and might have suffered from seemingly, but not actually, converged output-modules. This might also be solvable by simply letting the network train a couple of cycles after convergence, by the used criteria, of course, is reached.

bf Use larger computers. This also seems to go for CALMs. Even though runs after only a couple of generations seemed to get stuck on a plateau, often enough improvements (slight as they may have been) were found after more than 100 generations. There is no way of telling whether letting a simulation run for, for instance, 5000 generations would not produce eventually a member with a fitness of 100 (although the chance is admittedly small), however letting a simulation run for 200 generations on an indigo already costs several days, and with the problem of computers going down regularly it is difficult to run long simulations. The last problem can be solved by saving data to disk after each generations, something that is not done at this moment.

Acknowledgements

This research started out on the works of Boers and Kuiper ([Boer 92]), Murre ([Murr 92]) and Happel ([Happ 92]). Because of that some parts of this thesis are adaptations from their work, at times meaning that the text offered by them was followed rather closely.

Also some of the figures in this thesis were taken from, or adaptations of figures found in [Boer 92]: 2.1, 3.1, 3.2, 4.2, 4.3(a), 4.4, 7.2, 7.3 and 9.14, in [Murr 92]: 5.1, 5.2 and 5.3 and in [Free 91]: 4.1.

Finally, the "Thank you" section: Bart, Ida and Egbert for guiding me, Marko and Lawrence for some good suggestions and brainstorming at times, Nikè (for wading through umpteens pages of gibberish), Jur, Kirsten, Leo, Bommel and all others who showed interest and gave support and escape me at the moment.

Appendix A

The Default Parameters Of CALM

This appendix shows one of the most aspects. Figure A.1 shows the names and functions of the parameters, figure A.2 shows the default values of the parameters. It should be noted that these parameters values do not work well for every type of network architecture, in fact, they do not work well for most. The parameter values can be changed per CALM.

In this research the default parameters were mostly used for determining more suitable values for the parameters: they were used as a basis for the evolutionary strategies.

name	function
UP	Up-weight
DOWN	Down-weight
CROSS	Cross-weight
FLAT	Flat-weight
HIGH	High-weight
LOW	Low-weight
AE	AE-weight
ER	ER-weight (strange-weight)
INIT_LWT	Initial learning weight
INIT_MU	Reset value
ACTIV_DECAY_PAR	Decay parameter activation rule
K_LEARNING_PAR	Grossberg K-parameter learning rule
L_LEARNING_PAR	Grossberg L-parameter learning rule
D_LEARNING_BASERATE	base rate learning rule
E2MU_VIRTUAL_WEIGHT	Virtual weight from e to mu
LOWCONVERGENCE_CRITERIUM	Low convergence criterium
HIGHCONVERGENCE_CRITERIUM	High convergence criterium
ERROR	Random excitation error
VE	Weight from V to E node
OUT_RESET	Weights from output module R-nodes to Reset-node in paired CALM module
ANODE_RESET	weights from A-node to Reset-node of paired output module
RESET_RESET	Weight from Reset-node in CALM-module to Reset-node in paired output module
VNODE_DECAY_IN_STM	Decay V-nodes in STM module
OUTPUT2STM	Weights from output module to STM-module (one to one)
CALM2OUTPUT	Weights from CALM module to output module (one to one)
RNODE_INHIBIT_ON_OUTPUT_MOD	Decay R-nodes in output
LATERAL_INHIBIT	Lateral inhibition V-nodes in STM-module
AUTO_DECAY	Auto decay of weights
THRESHOLD	Threshold for activation V-nodes in STM module
USEPAR2	User defined parameter
⋮	⋮
USEPAR9	User defined parameter

Figure A.1: The names and functions of the CALM-parameters.

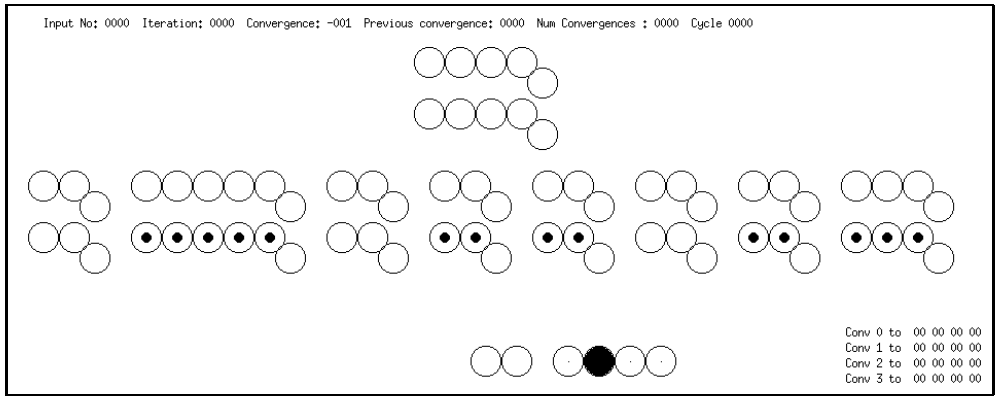
parameter number	name	value
0	UP	1.0
1	DOWN	-1.0
2	CROSS	-9.0
3	FLAT	-1.5
4	HIGH	-0.4
5	LOW	0.6
6	AE	1.4
7	ER	0.5
8	INIT_LWT	0.4
9	INIT_MU	0.005
10	ACTIV_DECAY_PAR	0.12
11	K_LEARNING_PAR	0.7
12	L_LEARNING_PAR	0.8
13	D_LEARNING_BASERATE	0.003
14	E2MU_VIRTUAL_WEIGHT	0.03
15	LOWCONVERGENCE_CRITERIUM	0.00001
16	HIGHCONVERGENCE_CRITERIUM	0.00010
17	ERROR	0.00005
18	VE	0.00000
19	OUT_RESET	1.0
20	ANODE_RESET	1.2
21	RESET_RESET	0.18
22	VNODE_DECAY_IN_STM	0.005
23	OUTPUT2STM	5.0
24	CALM2OUTPUT	3.0
25	RNODE_INHIBIT_ON_OUTPUT_MOD	0.25
26	LATERAL_INHIBIT	0.0
27	AUTO_DECAY	0.0
28	THRESHOLD	0.5
29	USEPAR2	
⋮	⋮	⋮
36	USEPAR9	

Figure A.2: The default values of the CALM-parameters.

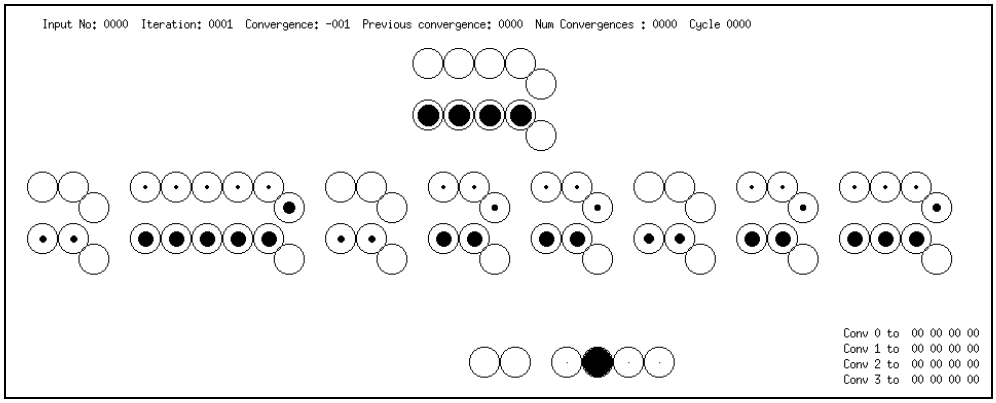
Appendix B

An Example Of Categorization In A CALM-network

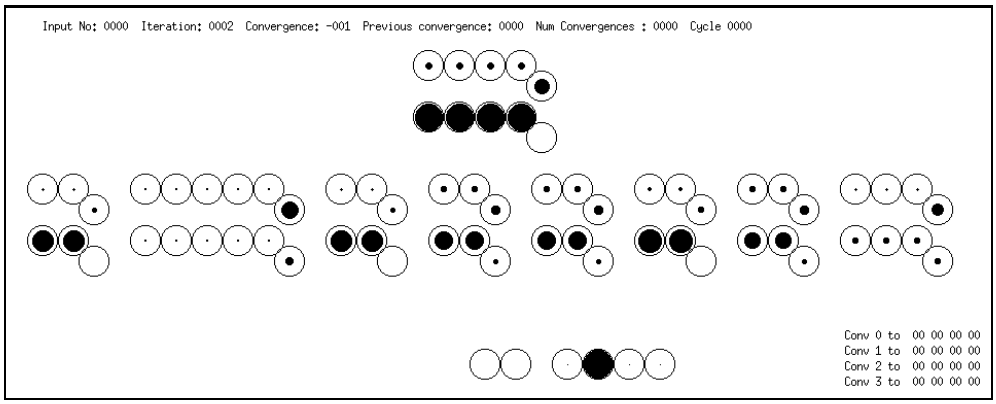
This appendix gives a rather large example of the categorization of one input by a CALM. Due to an unknown cause the exact architecture of the network was lost, but the figures were still added to this thesis because they still show some of the processes that rule within a CALM-network.



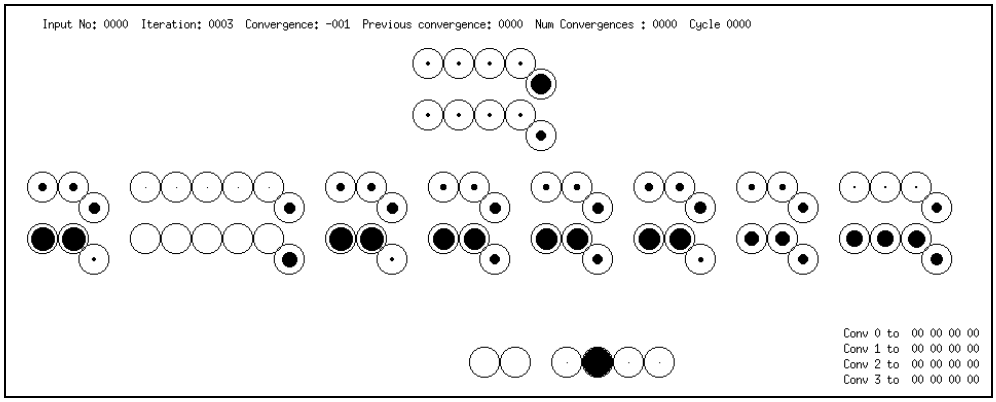
(a)


Input No: 0000 Iteration: 0001 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

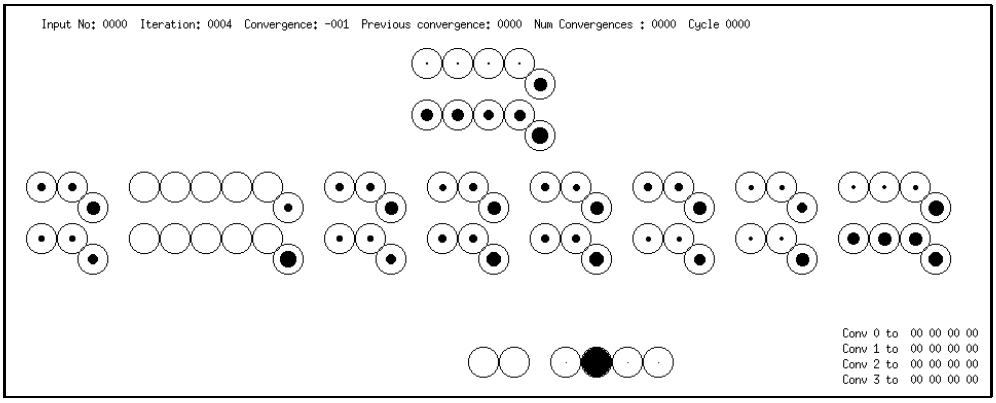
(b)


Input No: 0000 Iteration: 0002 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

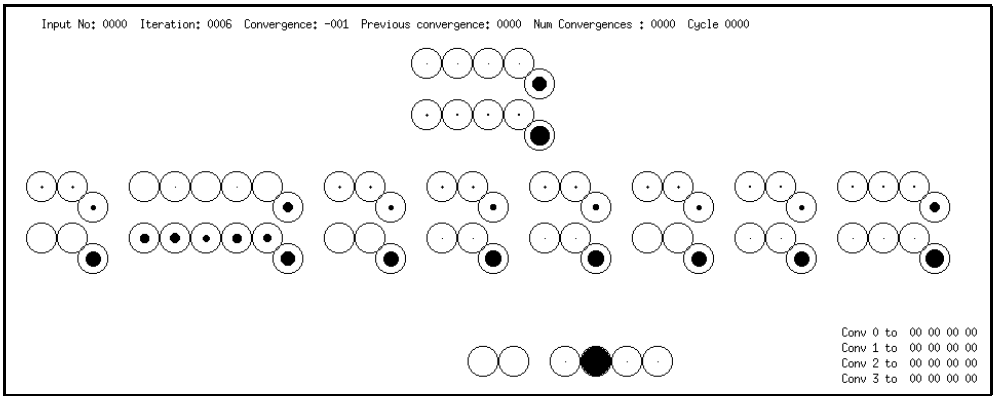
(c)


Input No: 0000 Iteration: 0003 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

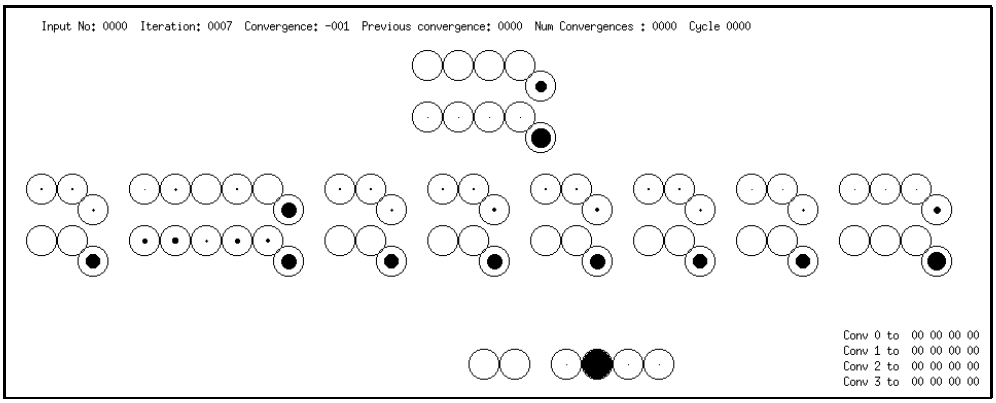
(d)


Input No: 0000 Iteration: 0004 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

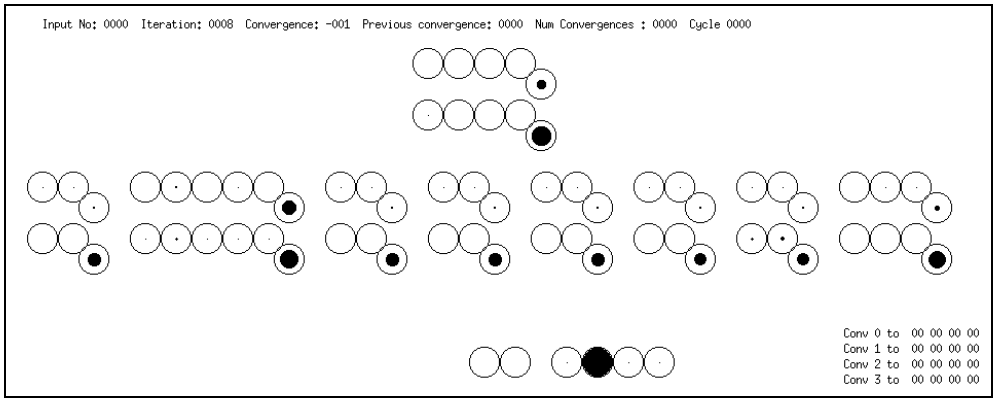
(e)


Input No: 0000 Iteration: 0006 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

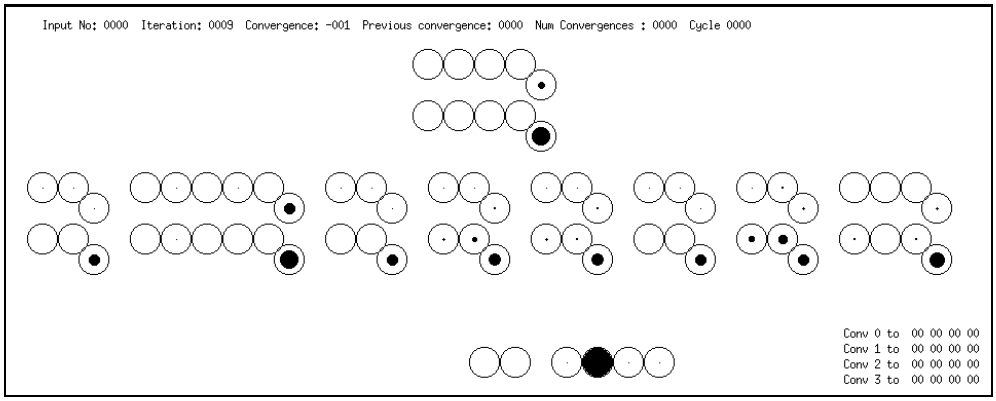
(f)


Input No: 0000 Iteration: 0007 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

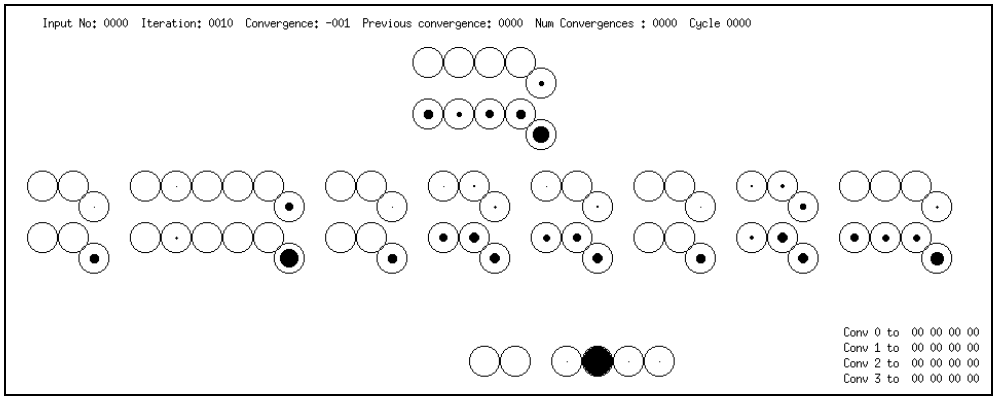
(g)


Input No: 0000 Iteration: 0008 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

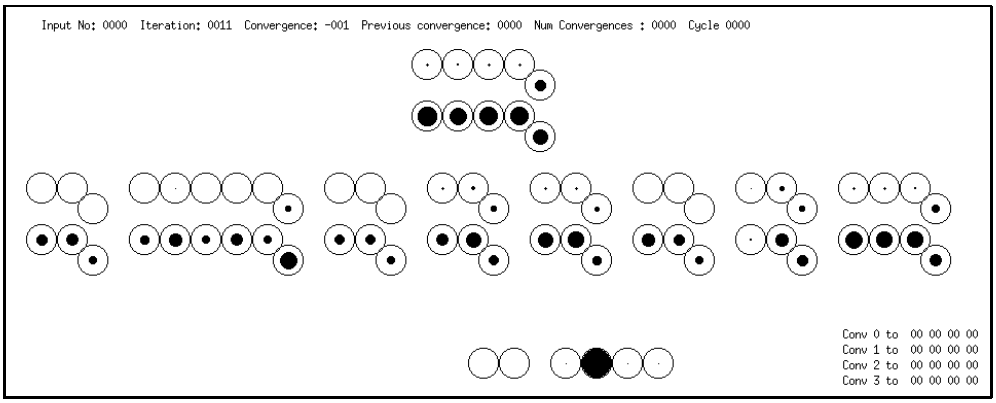
(h)


Input No: 0000 Iteration: 0009 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

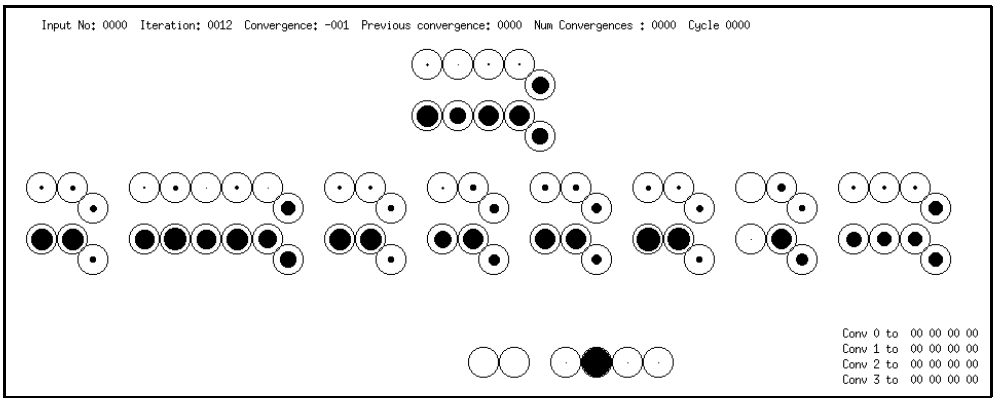
(i)


Input No: 0000 Iteration: 0010 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

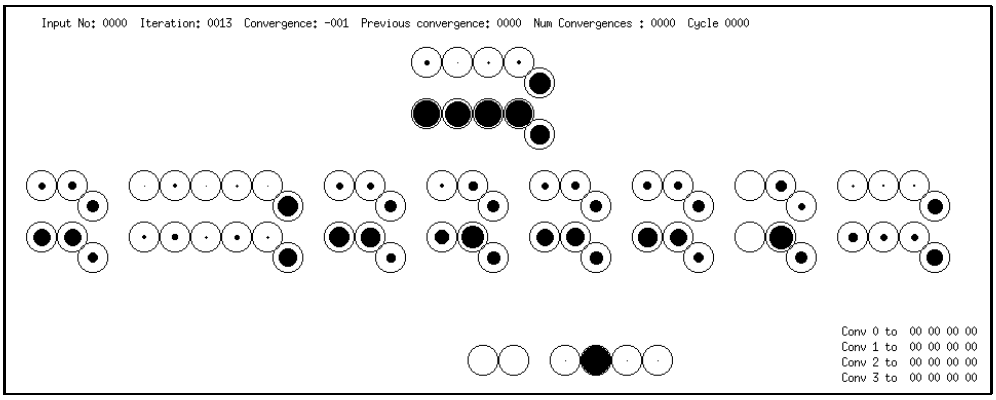
(j)


Input No: 0000 Iteration: 0011 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

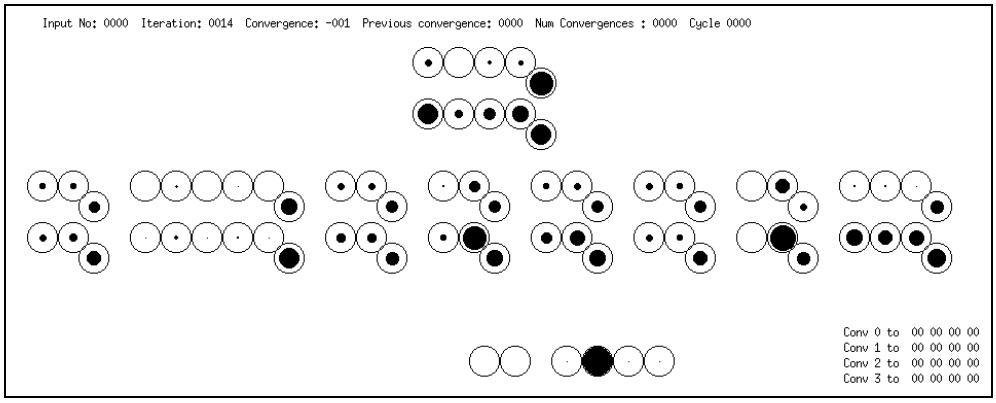
(k)


Input No: 0000 Iteration: 0012 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

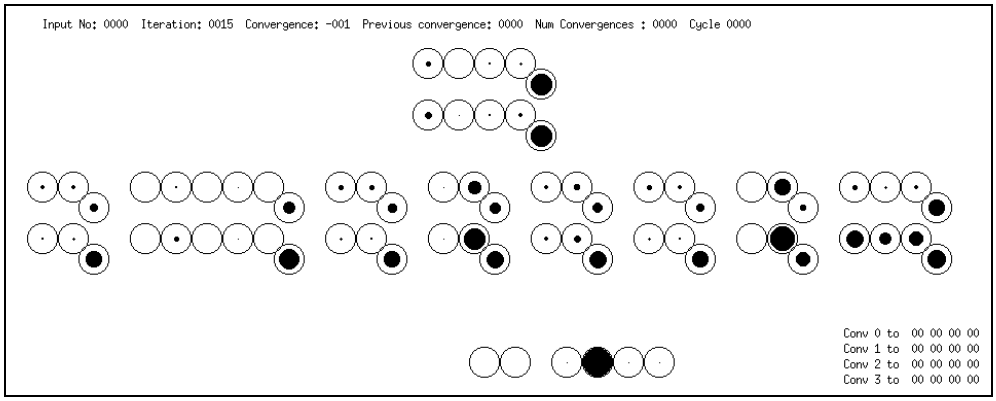
(l)


Input No: 0000 Iteration: 0013 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

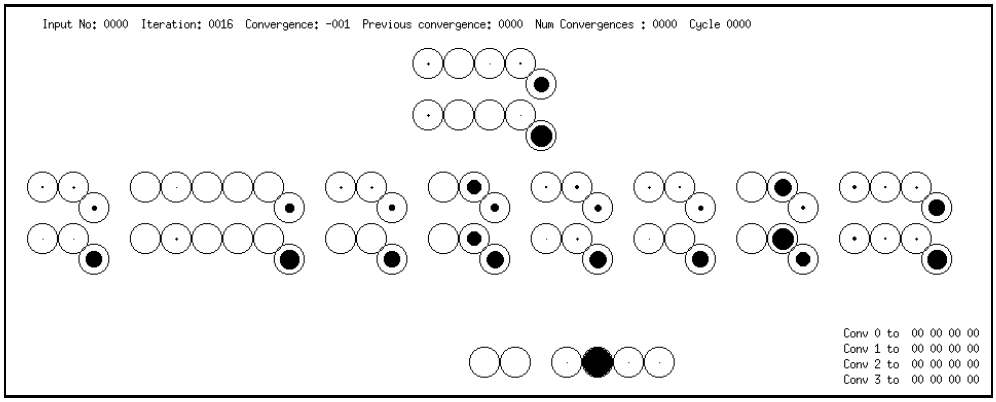
(m)


Input No: 0000 Iteration: 0014 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

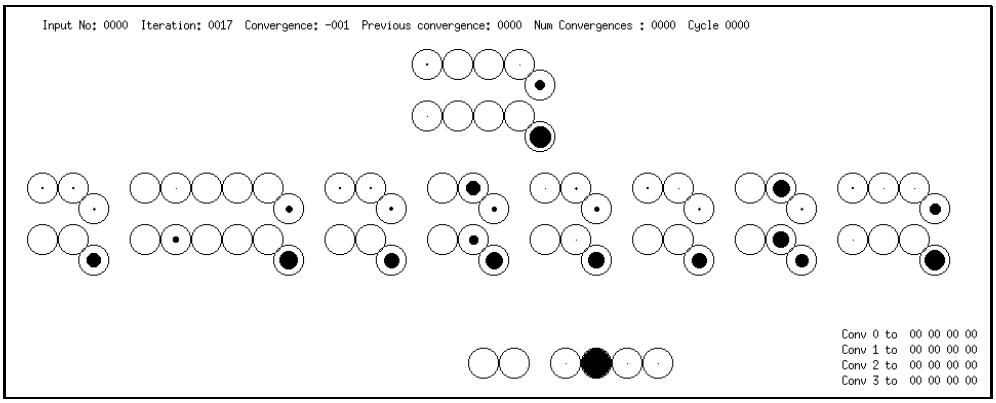
(n)


Input No: 0000 Iteration: 0015 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

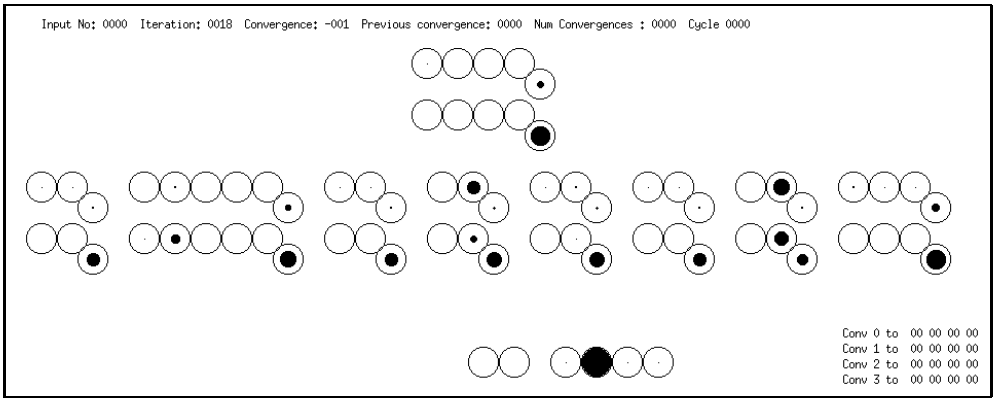
(o)


Input No: 0000 Iteration: 0016 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

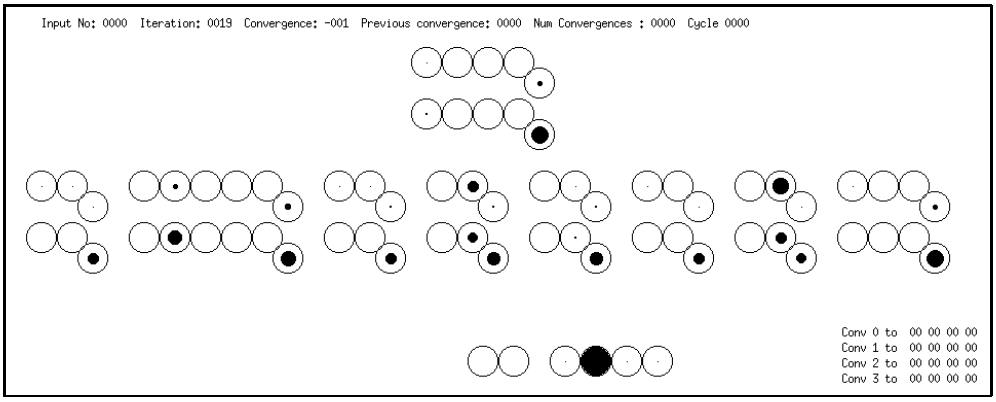
(p)


Input No: 0000 Iteration: 0017 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

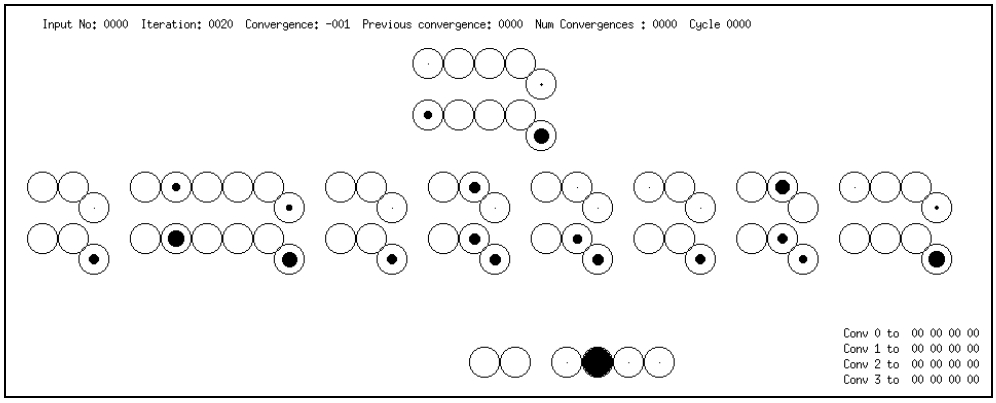
(q)


Input No: 0000 Iteration: 0018 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

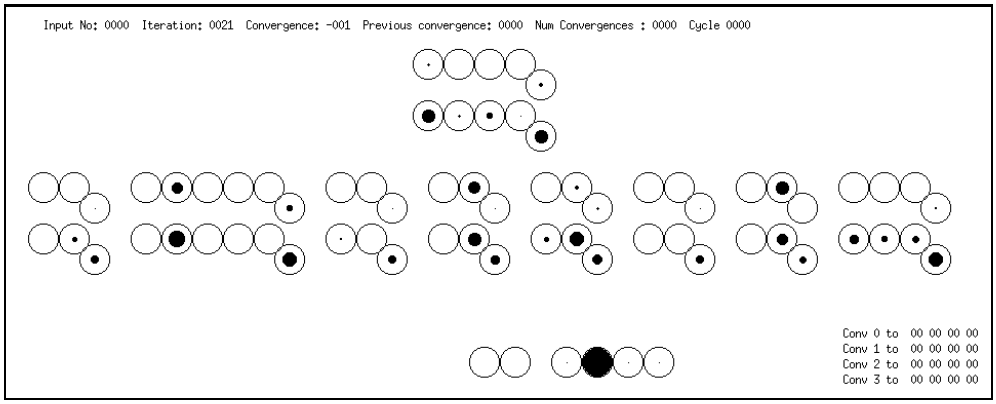
(r)


Input No: 0000 Iteration: 0019 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

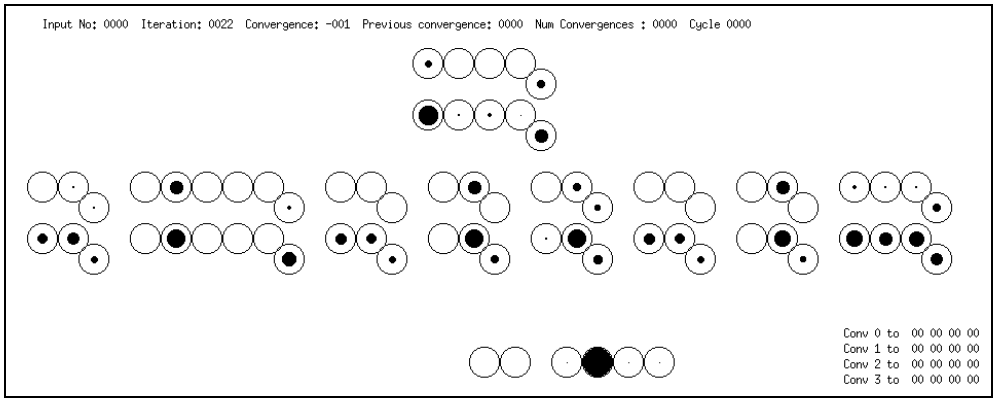
(s)


Input No: 0000 Iteration: 0020 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

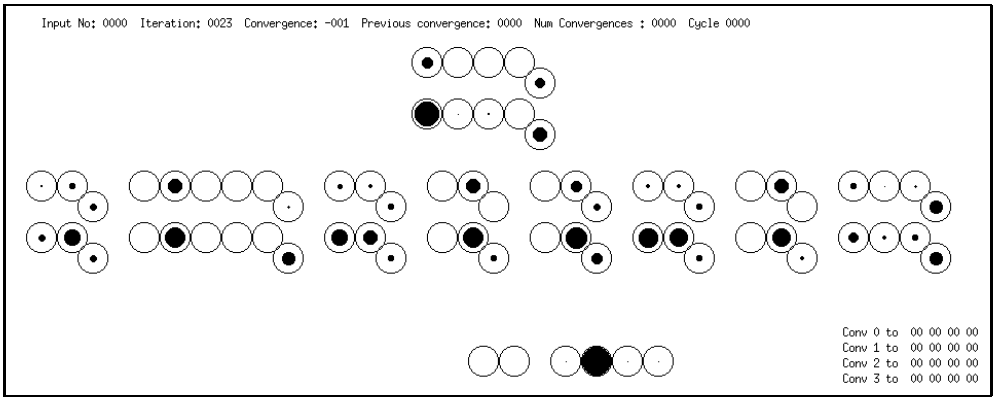
(t)


Input No: 0000 Iteration: 0021 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

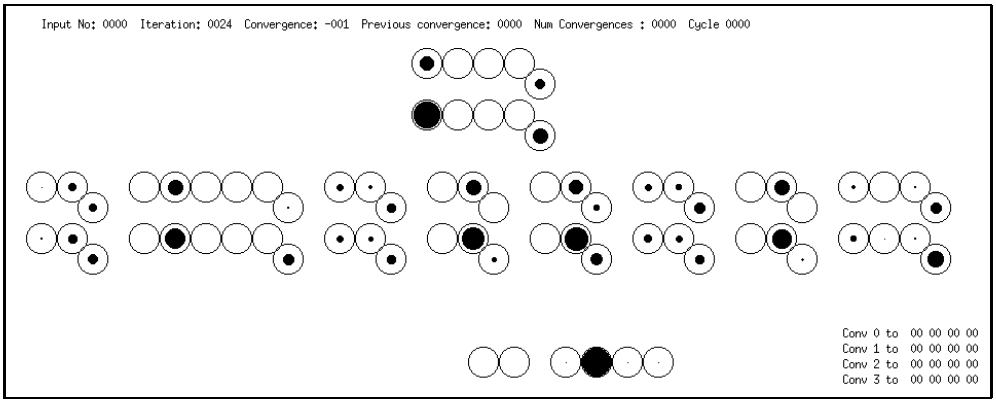
(u)


Input No: 0000 Iteration: 0022 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

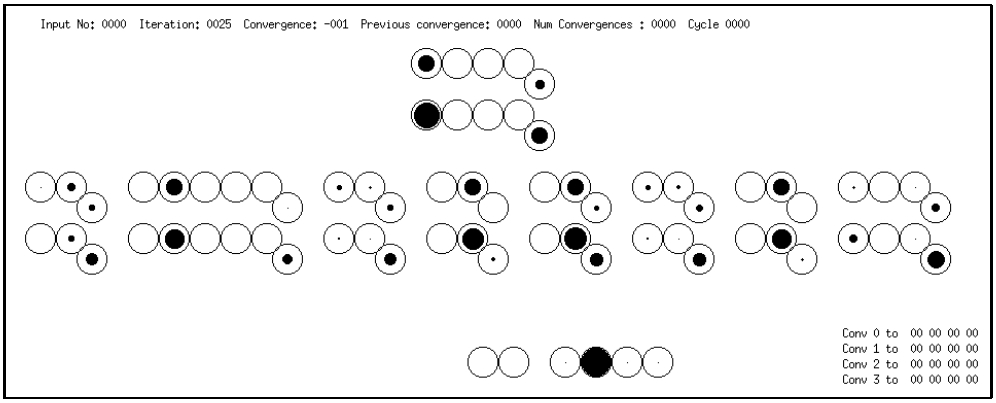
(v)


Input No: 0000 Iteration: 0023 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

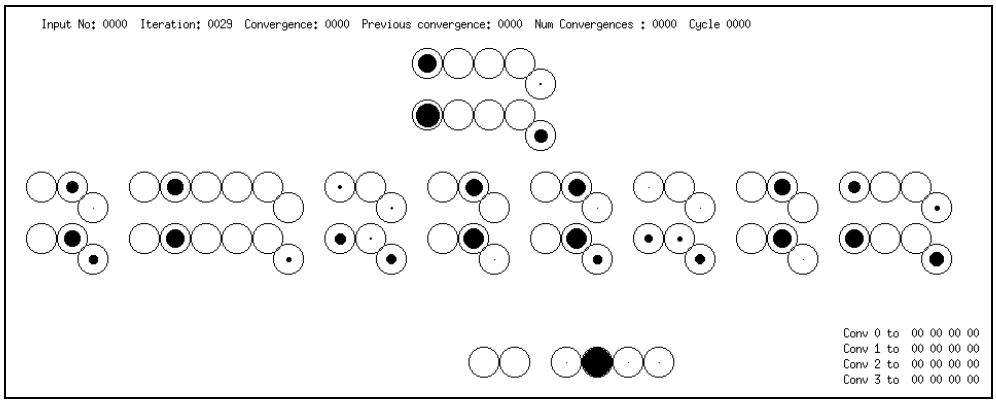
(w)


Input No: 0000 Iteration: 0024 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

(x)


Input No: 0000 Iteration: 0025 Convergence: -001 Previous convergence: 0000 Num Convergences : 0000 Cycle 0000
Conv 0 to 00 00 00 00
Conv 1 to 00 00 00 00
Conv 2 to 00 00 00 00
Conv 3 to 00 00 00 00

(y)



(z)

Figure B.1: This figure shows the process of an unlearned network categorizing offered input. This is the first time input is offered to the network, so all weights are at initialization value. (a) shows how all R-nodes in most CALMs get activated, whilst (b) shows how this input is propagated to the output CALM as well as the upcoming of activation in the V-nodes in the internal CALMs. The high activations are typically caused by unknown patterns. (c) shows how some of the A-nodes get activated because of the strong competition of the R-nodes, caused by the unknown pattern, as well a lot of R-nodes getting less active because of the strong inhibition by the V-nodes (which also get activated through the strong competition). (d) and (e) show how some R-nodes pick up activation for the second time, a process known as oscillation, one of the features of CALM-networks. (f) through (i) shows the first signs of the network getting familiar with the pattern: not all R-nodes in a CALM have the same activation, and activations change less rapidly, although (j) through (l) show the network still has some work to do: a lot of nodes are still able to reach a high activation (also in the output module, which should in the end have just one R-node with a significant activation). (m) through (o) show that the differentiation between activations within one CALM grow: there are CALMs with virtually unactivated nodes as well as strongly activated nodes (typically what we want to have in categorization), the differences in the activations in the output module already hint at the node to which the input is categorized. (p) through (r) show the network in the lull before the final run to categorization. These lulls become gradually longer with the learning of the pattern. (s) through (y) show how only a limited number of R-nodes get activated now that the pattern is learned, the learning of the problem seems not only to result in the output module having just one activated R-node (which in fact is the most common criterion for determining whether the pattern is learned), but also in most of the internal modules having just one activated R-node. This in principle limits the calculative powers of the network: so much less patterns might be coded when just one node is activated, compared to combinations of activated nodes. (z) shows the final activations: just one R-node is activated, activation in the others in negligible: the pattern is learned.

Bibliography

[Allp 80] D.A. Allport; "Pattern and Actions". In: *New Directions in Cognitive Psychology*, C.L. Clagton (Ed.), Routledge and Kegan Paul, London, 1980.

[Bem 89] S. Bem, *Het Bewustzijn Te Lijf:* een Geschiedenis van de Psychologie. Boom Meppel, Amsterdam, 1989.

[Bloo 88] F. Bloom and A. Lazerson; *Brain, Mind, and Behavior.* Freeman, 1988.

[Boer 92] E.J.W. Boers and H. Kuiper; *Biological Metaphors And The Design Of Modular Artificial Neural Networks.* Department of Computer Science, University of Leiden, 1992.

[Boer 95] E.J.W. Boers; *Using L-Systems As Graph Grammars: G2L-Systems.* Department of Computer Science, University of Leiden, 1995.

[Carp 87] G.A. Carpenter and S. Grossberg; Neural Dynamics of Category Learning and Recognition: Attention, Memory Consolidation, and Amnesia. In: J. Davis, R. Newburgh and E. Wegman (Eds.) *Brain Structure, Learning, and Memory.* AAAS Symposium Series, 1987.

[Darw 1859] C. Darwin; *The Origin of Species.* John Murray, 1859.

[Dawk 86] R. Dawkins; *The Blind Watchmaker.* Longman, 1986. Reprinted with appendix by Penguin, London 1991.

[Free 91] J.A. Freeman and D.M. Skapura; *Neural Networks: Algorithms, Applications and Programming Techniques.* Addison-Wesley, Reading, 1991.

[Gari 90] H. De Garis; "Brain Building with GenNets". In: *Proceedings of the International Neural Network Conference, INNC-90-Paris*, 1036-1039, B. Widrow and B. Angeniol (Eds.), Kluwer, Dordrecht, 1990.

[Gazz 89] M.S. Gazzaniga ; "Organization Of The Human Brain". In: *Science, 245*, 947-952, 1989.

[Gold 89] D.E. Goldberg; *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, 1989.

[Gros 76] S. Grossberg; "Adaptive Pattern Classification and Universal Recoding, II: Feedback, Expectation, Olfaction, and Illusions". In: *Biological Cybernetics, 23*, 187-202, 1976.

[Guyt 86] A.C. Guyton; *Textbook of Medical Physiology.* Saunders, Philadelphia, 1986.

[Happ 92] B.L.M. Happel; *Architecture and Function of Neural Networks: Designing Modular Architectures.* Department of Psychology, University of Leiden, In preparation 1992.

[Hech 90] Robert Hecht-Nielsen; *Neurocomputing.* Addison-Wesley, 1990.

[Hill 93] W.D. Hillis and L.W. Tucker; "The CM-5 Connection Machine: a Scalable Supercomputer". In: *Communications of ACM, Vol.36, No.11*, 31-40, 1993.

[Hoge 74] P. Hogeweg and B. Hesper; "A Model Study on Biomorphological Description". In: *Pattern Recognition, 6* 165-179, 1974.

[Holl 75] J. Holland; *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[Hoog 91] R.J.W. van Hoogstraten; *A Neural Network for Genetic Facies Recognition.* Unpublished student report, Leiden, 1991.

[Hube 62] D.H. Hubel and T.N. Wiesel; "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex". In: *Journal of Physiology, 160*, 106-154, 1962.

[Kand 85] E.R. Kandel and J.H. Schwartz; *Principles of Neuroscience.* Elsevier, New York, 1985.

[Lind 68] A. Lindenmayer; "Mathematical Models for Cellular Interaction in Development, parts I and II". In: *Journal of Theoretical Biology, 18*, 280-315, 1968.

[Murr 92] J.M.J. Murre; *Categorization And Learning In Neural Networks* Modelling and Implementation in a Modular Framework. Ph.D. thesis RUL, januari 1992.

[Norr 90] Norris; "How to Build a Connectionist Idiot (Savant)". In: *Cognition, 35*, 277-291, 1990.

[Park 85] D.B. Parker; *Learning Logic.* MIT Press, Cambridge, MA, 1985

[Prus 89] P. Prusinkiewicz and J. Hanan; *Lindenmayer Systems, Fractals and Plants.* Springer-Verlag, New York, 1990.

[Ruec 89] J.G. Rueckl, K.R. Cave, and S.M. Kosslyn, "Why Are 'what' and 'where' Processed by Separate Cortical Visual Systems? A Computational Investigation". In: *Journal of Cognitive Neuroscience, 1*, 171-186, 1989

[Rume 86] D.E. Rumelhart and J.L. McClelland; *Parallel Distributed Processing. Volume 1: Foundations.* MIT Press, Cambridge, MA, 1986.

[Schw 88] J.T. Schwartz; *The New Connectionism: Developing Relationships between Neuroscience and Artificial Intelligence*, 1988.

[Szil 79] A.L. Szilard and R.E. Quinton; "An Interpretation for D0L-systems by Computer Graphics". In: *The Science Terrapin, 4*, 8-13, 1979.

[Turi 63] A. Turing; "Computing Machinery and Intelligence". In: *Computers and Thought*, E.A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.

[Warr 70] E.K. Warrington and L. Weiskrantz; "Amnesia Syndrome: Consolidation or Retrieval?". In: *Nature, 228*, 628-630, 1970.

[Werb 74] P.J. Werbos; *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Unpublished Ph.D. thesis, Harvard University, Cambridge, MA, 1974.

[Whit 89] D. Whitley; "The GENITOR Algorithm and Selection Pressure: Why Rank-based Allocation of Reproductive Trials Is Best". In: *Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications (ICGA)*, 116-1212, J.D.Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.