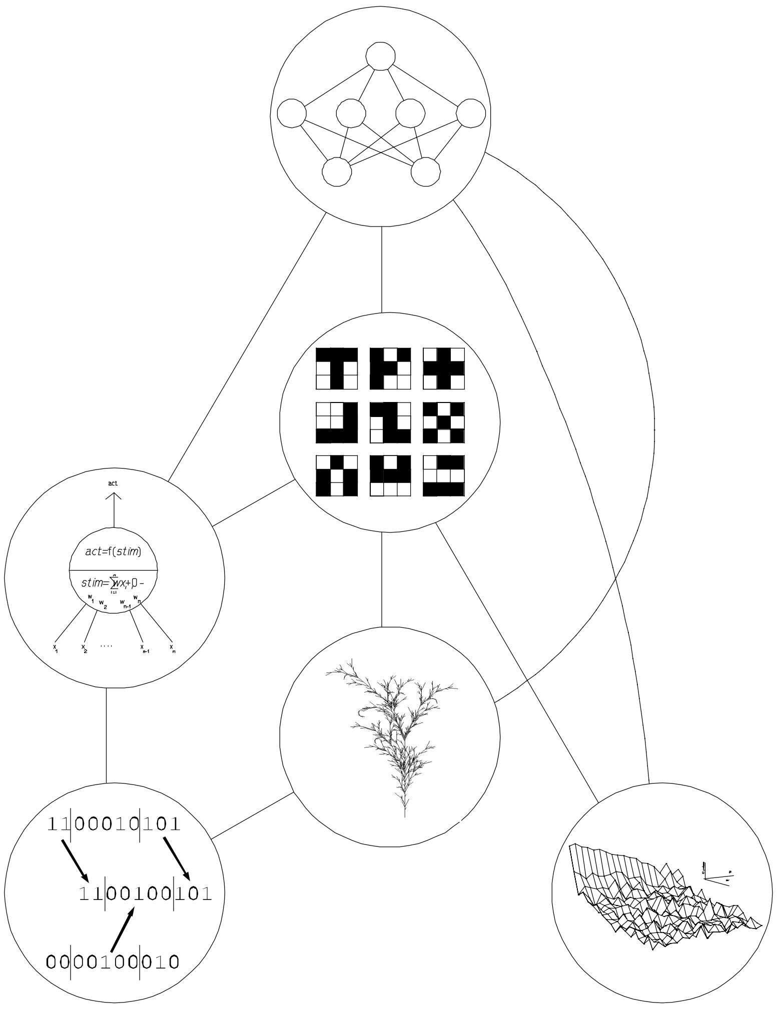
Biological metaphors and the design of modular artificial neural networks



Master's thesis of Egbert J.W. Boers and Herman Kuiper

Departments of Computer Science and
Experimental and Theoretical Psychology
at Leiden University, the Netherlands

Preface

This thesis is the result of a research done at the departments of Computer Science and Experimental and Theoretical Psychology at Leiden University, the Netherlands. It is a part of the project ‘Architecture and Function of Modular Neural Networks’, done at the department of Experimental and Theoretical Psychology by Bart L.M. Happel[1].

The research comprised four parts: first, extensive reading was done to get familiar with the areas of neural networks, genetic algorithms and formal grammars. Secondly, new methods were developed to design modular structures, which were then transformed into software with which a number of experiments were done. Finally, this thesis was written, together with manuals for the software developed.

We wish to thank Bart Happel for his stimulating input and excellent suggestions whenever we got stuck. Also, we wish to thank Ida Sprinkhuizen-Kuyper for her guidance throughout the project. Finally, we thank the numerous proofreaders who helped to write (hopefully) a faultless thesis and all the Sun users at the Computer Science department who had to put up with our simulations.

Leiden, august 1992

Egbert Boers
Herman Kuiper

[1] *His work is sponsored in part by the Dutch Foundation for Neural Networks.*

Abstract

In this thesis, a method is proposed with which good modular artificial neural network structures can be found automatically using a computer program. A number of biological metaphors are incorporated in the method. It will be argued that *modular* artificial neural networks have a better performance than their non-modular counterparts. The human brain can also be seen as a modular neural network, and the proposed search method is based on the natural process that resulted in the brain: Genetic algorithms are used to imitate evolution, and L-systems are used to model the kind of *recipes* nature uses in biological growth.

A small number of experiments have been done to investigate the possibilities of the method. Preliminary results show that the method *does* find modular networks, and that those networks *outperform* 'standard' solutions. The method looks very promising, although the experiments done were too limited to draw any general conclusions. One drawback is the large amount of computing time needed to evaluate the quality of a population member, and therefore in chapter 9 a number of possible improvements are given on how to increase the speed of the method, as well as a number of suggestions on how to continue from here.

Contents

'Your quote could have been here.'

1 Introduction

Modularity[1] *is everywhere*. From astrophysics to quantum mechanics, one can see modularity. In biology, the growth and development of living organisms is modular. The human brain also is probably highly modular. Modularity in the brain can be identified at many different levels, physical as well as functional. In this thesis, it will be argued that when modelling the brain using *artificial neural networks* in order to create so-called *intelligent* software, exploiting modular design principles will result in better networks. Also, it will be shown that using techniques based on biological *genetics* and *growth* to search for optimal neural network topologies can result in an all-round search for good topologies for a variety of problems.

Research goals

Preliminary results show that using modularity when designing artificial neural networks can improve their performance. As will be set forth in chapter 2, the topology of a network has a large influence on the performance of that network but, so far, no method exists to determine the optimal topology for a given problem because of the high complexity of large networks. The (human) brain is a large scale neural network based on modularity and it might therefore be a good idea to do a *reverse engineering* of the genetic search and development processes that led to the brain. The genetic search in nature resulted in the usage of a kind of *recipes* (instead of *blueprints*) to describe the development of the organism.

The goal of this research was to develop a method with which good neural network topologies could be found using a computer. Because of the excellent results found in nature, we strived to keep our method as *biological plausible* as possible, so the found topologies should have a

[1] In this thesis, modularity is defined as a subdivision in identifiable parts, each with its own purpose or function.

high degree of modularity and should be found using genetic search and recipes instead of blueprints. L-systems (which can be seen as a kind of recipe) are able to encode repeated patterns (modules) with a complex internal structure efficiently. Genetic algorithms (used to simulate evolution) and L-systems are treated in chapters 3 and 4. Chapter 5 provides a more thorough treatment of modularity in nature and also discusses the usefulness of this principle for our research.

The thesis is divided in three parts: in the first part (chapters 2 through 4) introductions are given to the three main disciplines used in this research. The second part (chapters 5 through 7), gives a more thorough treatment of the ideas behind this research and the software developed. Finally, in the last part, (chapters 8 and 9), the results of a number of experiments are reported and recommendations for further research are given. In the remainder of this chapter the used disciplines are briefly introduced.

Neural networks

Ever since computers were invented, humans have tried to imitate (human) intelligent behaviour with computer programs. This is not an easy task because a computer program must be able to do many different things in order to be called intelligent. Also, the meaning of the word *intelligence* is somewhat unclear, because many different definitions exist.

The methods used to achieve artificial intelligence in the early days of computers, like rule based systems, never achieved the results expected and so far it has not been possible to construct a set of rules that is capable of intelligence. Because *reverse engineering* proved to be successful in many other areas, researchers have been trying to model the human brain using computers. Although the main components of the brain, *neurons*, are relatively easy to describe, it is still impossible to make an artificial brain that imitates the human brain in all its detailed complexity. This is because of the large numbers of neurons involved and the huge amount of connections between those neurons. Therefore large simplifications have to be made to keep the needed computing power within realistic bounds.

An artificial neural network consists of a number of *nodes* which are connected to each other. Each of the nodes receives input from other nodes and, using this input, calculates an output which is propagated to other nodes. A number of these nodes are designated as *input* nodes (and receive no input from other nodes) and a number of them are designated as *output* nodes (and do not propagate their output to other nodes). The input and output nodes are the means of the network to communicate with the outside world.

There are a number of ways to *train* the network in order to learn a specific problem. With the method used in this research, *backpropagation*, *supervised learning* is used to train the network. With supervised learning, so-called *input/output pairs* are repeatedly presented to the net. Each pair specifies an input value and the output that the network is supposed to produce for that input. To achieve an internal representation that results in the wanted input/output behaviour, the input values are propagated through the nodes. Using the difference between the resulting output and the desired output, an error is calculated for each of the output nodes. Using these error values, the internal connections between the nodes are adjusted. This process is described in detail in chapter 2.

Genetic algorithms

Darwinian evolution, where fit organisms are more likely to stay alive and reproduce than non-fit organisms, is modelled by *genetic algorithms*. A population of strings is manipulated, where each string can be seen as a *chromosome*, consisting of a number of *genes*. These genes are used to code the parameters for a problem for which a solution has to be found. Each string can be assigned a *fitness*, which indicates how good the string is as solution for the problem. As with natural selection and genetics, where the chance of reproduction for an organism (and thus its genes) depends on its ability to survive (its fitness), the strings used by the algorithm reproduce proportional to their fitness. A new generation is created by selecting and recombining existing strings based on pay-off information (their fitness) using a number of genetic operators. The most commonly used operators are *selection*, *crossover*, *inversion* and *mutation*. They are described in detail in chapter 3. In that chapter also a more thorough treatment is provided of the basic mechanisms of genetic search as implemented by a genetic algorithm.

L-systems

The development of living beings is governed by genes. Each living cell contains genetic information (the *genotype*) which determines the way in which final form of the organism will develop (the *phenotype*). This genetic information is not a blueprint of that final form, but can be seen as a *recipe*. This recipe is followed not by the organism as a whole, but by each cell individually. The shape and behaviour of each cell depends on the genes from which information is extracted and this in turn depends on which genes have been read in the past and on influences from the environment of all the neighbouring cells. So the development is solely governed by the local interactions between elements that obey the same global rules. In order to model this development in plants, the biologist Aristid Lindenmayer developed a mathematical construct, called *L-systems*. By using so-called *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. The application of a rewriting rule depends on which rules have been applied in the past and on the neighbouring characters of the character to be rewritten. Chapters 4 and 6 describe the standard L-systems and the L-system used during this research.

Overview

The method that resulted from the combination of the three techniques can be described as follows. When looking for a network structure that is able to learn a task, a genetic algorithm is used to produce production rules for an L-system. These rules result in a network structure. That structure can then be evaluated by looking at the extent in which the network can learn the task. This results in a fitness which is coupled to the original genetic coding of the production rules. The fitness enables the genetic algorithm to evolve a set of production rules that generate an optimal network structure.

‘*The human brain is like an enormous fish. It is flat and slimy and it has gills through which it can see. Should one of these gills fail to open, the messages transmitted by the lungs don’t reach the brain. It is as simple as that.*’

— Monty Python on a disease called *hooping cough*

2 Neural Networks

‘*We animals are the most complicated things in the known universe.*’ This quote from the biologist Richard Dawkins [DAWK86] (p.1) assents the complexity of the problems seen in biology. The brain is perhaps the most complex organ of an animal, particularly in humans. Discovering how the brain works, and how it is able to be intelligent, is probably the most challenging task ever.

The human brain

The human brain is a network of a huge number of interconnected *neurons*. Each neuron has a very complex bio-electrical and bio-chemical behaviour, though its basic computational principles are believed to be very simple: it adds its input and performs a threshold operation. In other words: every neuron is capable of adjusting its output as a relatively simple function of its input. How can it be that such a rather simple basic unit is able to generate such a complex behaviour? The main key to the answer is the cooperation and interaction between neurons. Although they work relatively slow compared to modern computers, there are a lot of them and they all operate in parallel. This combined effort of these large numbers of neurons is what is believed to be the origin of human intelligence. What intelligence means, and what it *is*, is still unknown. But so far we humans seem to be the only ones who, in the future, may be able to provide the answer[1]. There is little doubt that if we discover the origin of intelligence, we will owe it to the wonderful organ in our head.

[1] An interesting question is whether the human brain is capable to understand its own functioning. Gödel’s theorem suggests that there may be ‘ideas’ which can not be understood by the brain, assuming the brain can be described as a formal system... (see e.g. [HOFS79] and [PENR89])

Artificial intelligence

The artificial intelligence community has for a long time been trying to imitate intelligent behaviour with computer programs. This is not an easy task because a computer program must be able to do many different things in order to be called intelligent. Something that has caused a lot of confusion is the definition of artificial intelligence: the Webster's Dictionary alone, for example, gives 4 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 to 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.

Allan Turing has proposed a test that should be satisfied in order to speak of artificial intelligence. In this test, known as the *Turing Test* [TURI63], a person, say Q, is placed in a room with a terminal connected to two other sites. At one of the two terminals a person is situated and at the other a computer. By asking questions, Q must determine at which of the two terminals the computer is situated. Turing is willing to accept the computer program as being intelligent if Q fails. Of course the set-up of the test should make it impossible to decide who is who by measuring response-time etcetera. The Turing Test is disputed because some believe it is possible to deceive Q without having an intelligent program.

The traditional ways of designing intelligent systems, like rule-based systems, never achieved the results that were expected at the time people started to realize that computers could be used for more than just calculating numbers. So far it has not been possible to construct a set of rules that is capable of intelligence. There are some expert systems able to compete on a specialist-level in very narrow areas, but there is no program yet that is capable of functioning in everyday situations.

The problems encountered with the more traditional methods urge more and more researchers to look for other approaches. A principle that has proved to be effective on many other occasions is *reverse engineering* (looking for something that works, trying to understand it and then rebuild it). In this particular case, it means looking at the natural brain and using its processing principles in a computer program.

The neuron

The *neurons* (the nerve cells) in the brain are the cells responsible for our thinking. Because all artificial neural networks are based on the way neurons work, it is important to have an idea of their functioning. The neuron, as shown in figure 1, can be divided in three functional parts: the *dendrites*, the *body* and an *axon*. The dendrites are the information collectors of the neuron. They receive signals from other neurons, and transmit those signals in the form of electrical impulses to the cell body. The body collects all these impulses and if the summed charge of all these impulses exceeds a certain threshold, the cell body activates the axon. The axon transmits this

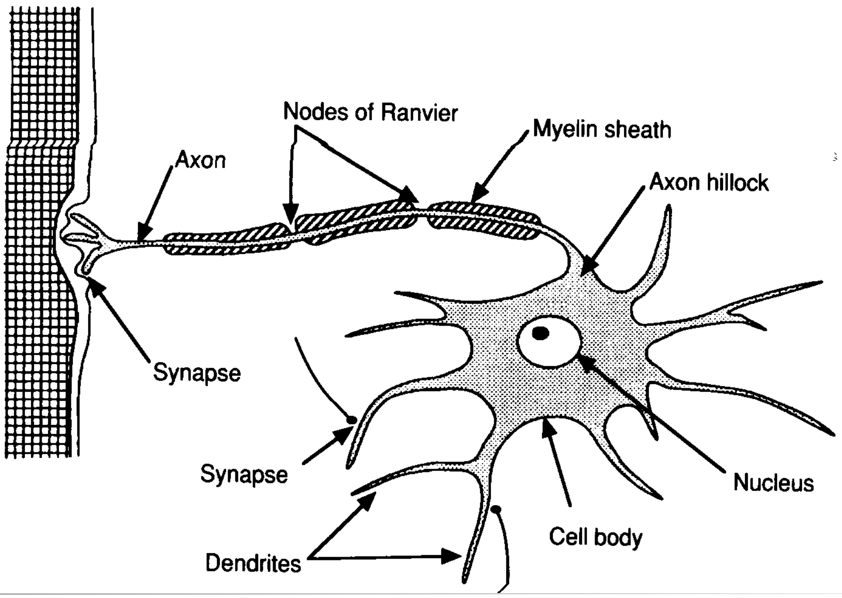


Figure 1. *The neuron.*

activation, again as an electrical impulse, to the dendrites of other cells. To enable the brain to change its internal processing and learn, the influence of one neuron to another is thought to be variable. The learning takes place at the junction of the axons and dendrites. Each axon splits into a number of these junctions, called *synapses*. Each synapse responds to the electrical impulse in an axon by releasing certain *neurotransmitters*. These chemical substances, when reaching the dendrite, cause the electrical activity that is collected at the body of the receiving cell. Changes in this process causes learning and development.

Neurons connected: the brain

The brain is made up of around 10^{11} neurons. It should be clear that it is quite impossible to fully connect each neuron with each other neuron. It has been estimated that a full connectivity would result in a head with a 10 kilometre diameter, because of the huge amount of wiring [HEEM91]. In order to reduce the amount of connections, the brain is divided in different modules at several levels. The clearest division of the brain is the division in a right and a left half, which function to a large extent independently. Instead of being fully connected, they are connected by a relatively small amount of connections through a structure called the *corpus callosum*. At a smaller scale the brain is divided in a number of *functional areas*, for example the visual area, auditory area, and separated sensory and motor areas, and so on. Between these areas too, only a relatively small number of connections exist. Gazzaniga [GAZZ89] describes a patient who was not able to name the colour of *red fruit*, after suffering from a head injury. The patient was able to name the colour of every other object presented to him, including other red objects. But when presented with red *fruit*, the answers where random. Apparently the specific connections between the areas where fruit and the colour red are recognized were lost. This example suggests a strong modularity in the brain, even at a smaller scale than the functional areas. Indeed, a lot of psychological and physiological research indicates a very strong modularization of the brain (see chapter 5). Despite of this highly specific structure of the brain, there still are a lot of connections between neurons. On average every neuron receives input from about 10 thousand synapses.

Because of these huge numbers of neurons and connections, it is clear why researchers have not been able to do a computer simulation of the brain, for the amount of computation needed is huge. Let us make an estimate of the computing power of the brain. Every axon is able to transmit one pulse every 10 milliseconds, and because the synapses are unable to use the differences in the amplitude of the impulse, the axon can be seen as a cable transmitting 100 bits per second. Combined with the total number of axons, this results in roughly 10^{13} bits per second. This is an estimation of just the data transmission in the brain: the amount of computation is even more staggering. Jacob Schwartz [SCHW88] estimates the total amount of arithmetic operations needed to simulate the brain in every detail as high as 10^{18} per second, needing 10^{16} bytes of memory. This is probably a million times as fast as the fastest supercomputer available in the next decade. It is also a motivation to do a lot of research in *massive parallel computers*.

Artificial neurons

As a consequence of the huge complexity of the human brain and the state of current hardware technology, 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.

Artificial neurons take their input (real numbers), and determine their output as a function of their input. Most of the time the total stimulation of these *processing elements* is simply the sum of all individual inputs multiplied by their corresponding weights. Sometimes a bias term θ is added in order to shift the sum relative to the origin: $stim = \sum_{i=1}^{n} w_i x_i + \theta$.

The analogy with the real neuron is obvious: in the brain the activation of a neuron is transmitted through the axon and arrives at the dendrites of the other cells where the strength of the synapses determines the extent in which the other cells are stimulated. Although real neurons give either excitatory or inhibitory signals at their synapses, in our model the weights of one node can be positive and negative. There is a clear distinction between the stimulation of a processing element and its activation, the latter is usually implemented as a function of the first:

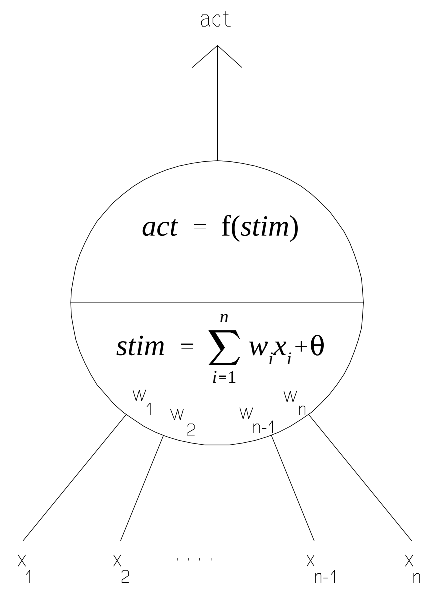


Figure 2. *The basic processing element.*

$$act = f(stim) = f\left(\sum_{i=1}^{n} w_i x_i + \theta\right).$$

Sometimes it is implemented as a function of the stimulation and the previous activation[1]:

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

[1] *Because of the digital simulation of the networks, generally time is considered to be measured in discrete steps. The notation t-1 indicates one timestep prior to time t.*

Although f is chosen depending on the kind of network that is being used, the basic functioning of the processing elements is the same in most neural networks, because it is assumed that this kind of artificial neuron implements the basic computational principles of the biologic neuron.

Artificial neural networks

As is the case with the brain, artificial neural networks are made up of a number of interconnected processing elements. The reader may now wonder what advantages neural networks offer compared to more traditional methods. One of the largest problems with almost all traditional artificial intelligence techniques is that the programmer has to supply all the knowledge (which is very often incomplete, unknown, or wrong). Even if a human expert ‘knows’ how to make certain decisions, for example in the medical world, he is very often incapable of telling exactly why and on what grounds those decisions are made. This is why a new area of research, called *knowledge engineering*, has arisen which purpose is to find methods for acquiring knowledge. Very often it happens that the knowledge collected cannot be described by definite rules, but has to be described using e.g. statistical reasoning methods and *fuzzy logic*. This difficulty of finding rules can be solved with neural networks, because they do not have to be told what to do, but are able to *learn* it by themselves. Neural networks are capable of autonomously discovering regularities, and extract knowledge from examples in complex task domains (like human experts). This is why the neural network approach is so promising, especially in areas lacking ‘absolute knowledge’.

The fact is that all traditional methods, like rule based systems, and even newer methods like fuzzy logic can be seen as special cases of neural networks. Since it is possible to construct the logical NAND function using a neural net even proves that *everything* a computer can do can be done using a neural network, because by combining these NAND networks a complete computer can be built. This shows how powerful artificial neural networks in the future will become, and even now there are already many examples available showing the strength of neural networks, although the neural network research is just at its beginning.

Some areas where neural networks are successfully used are (for a more exhaustive overview see e.g. [HECH90]):

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

but this list is far from complete, and new applications seem to appear every day.

Backpropagation

There are a lot of different neural network paradigms, of which backpropagation probably is the best known. It was formalized first by Werbos [WERB74] and later by Parker [PARK85] and

Rumelhart and McClelland [RUME86]. It is a *multi-layer feedforward* network that is trained by *supervised learning*. A standard back-propagation network consists of 3 layers, an input, an output and a hidden layer. The processing elements of both input and output layer are fully connected with the processing elements of the hidden layer, as shown in figure 3[1]. The fact that it is feedforward means that there are no recurrent loops in the network. The output of a node never returns at the same node, because cycles are not allowed in the network. In standard backpropagation this can never happen because the input for each processing element always comes from the previous layer (except the input layer, of course). This, again, is a large simplification compared with the real brain because the brain itself appears to contain many recurrent loops.

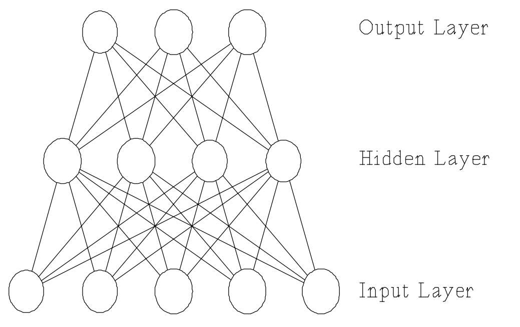


***Figure 3.** A typical backpropagation network.*

Supervised learning means that the network is repeatedly presented with input/output pairs `(I,O)` provided by a supervisor, where `O` is the output the network should produce when presented with input `I`. These input/output pairs specify the activation patterns of the input and output layer. The network has to find an internal representation that results in the wanted input/output behaviour. To achieve this, backpropagation uses a two-phase *propagate-adapt* cycle.

In the first phase the input is presented to the network and the activation of each of the nodes (processing elements) of the input layer is *propagated* to the hidden layer, where each node sums its input and propagates its calculated output to the next layer. The nodes in the output layer calculate their activations in the same way as the nodes in the hidden layer.

In the second phase, the output of the network is compared with the desired output given by the supervisor and for each output node the error is calculated. Then the error signals are transmitted to the hidden layer where for each node its contribution to the total error is calculated. Based on the error signals received, connection weights are then *adapted* by each node to cause the network to converge toward a state that allows all the training patterns (input/output pairs) to be encoded. For a more detailed description of backpropagation networks, the reader is referred to appendix A. A short description of the standard backpropagation algorithm for a network with *p* input, *q* hidden and *r* output nodes follows.

1. Initialize all the weights of the network with random values (e.g. between -1 and 1). We will denote the weights of the hidden layer and of the output layer as w_{ij}^{h} and w_{ij}^{o} respectively. The notation w_{ij}^{h} stands for the weight between input node i and hidden node j.

2. Choose an input/output pair $(\bar{x},\bar{y})$, where $\bar{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix}$ and $\bar{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_r \end{pmatrix}$ are the

[1] *Throughout this thesis, all connections in the figures point **upwards**. The input nodes are at the bottom, the output nodes at the top.*

inputvector and outputvector, and assign the inputvector to the corresponding input nodes.

3. Propagate the activation of the input layer to the hidden layer, and calculate the stimulation and activation of the hidden nodes. Often the bias (θ, see figure 2) of each node is implemented as an extra node 0 with a standard activation of 1, the weights from node 0 to the other nodes in the network are used as adaptive thresholds. The activation of the nodes in the hidden layer now becomes:

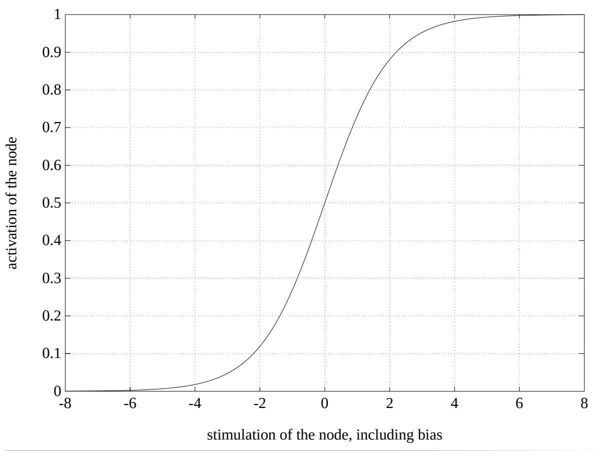


Figure 4. *The sigmoid function.*

$$h_j = \mathrm{f}(stim(h_j)) = \frac{1}{1+\mathrm{e}^{-\sum_{i=0}^{p} \mathrm{w}_{ij}^{h} \cdot \mathrm{x}_i}}.$$

f is the sigmoid function shown in figure 4.

4. Propagate the activation of the q hidden nodes to the output layer.

$$o_j = \frac{1}{1+\mathrm{e}^{-\sum_{i=0}^{q} \mathrm{w}_{ij}^{o} \mathrm{h}_i}}.$$

5. Calculate the deltas (the errors) of the output layer:

$$\delta_i^o = o_i(1-o_i)(y_i-o_i). \qquad \text{(see appendix A for a derivation)}$$

6. Compute the deltas for the hidden layer: $\delta_i^h = h_i(1-h_i)\sum_{j=1}^{r} \delta_j^o w_{ij}^o$.

7. Adjust the weights between the hidden layer and the output layer:

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

$$\text{where } \Delta w_{ij}^o(t) = w_{ij}^o(t+1) - w_{ij}^o(t).$$

The last term is called the *momentum*, it tends to keep the weight changes (Δw_{ij}) going in the same direction by averaging the changes over the last few training cycles. Usually α, the *learning-rate parameter*, is chosen between 0.1 and 0.5 and β, the *momentum parameter* between 0.8 and 0.95.

8. Adjust the weights between the input layer and the hidden layer:

$$w^h_{ij}(t+1) = w^h_{ij}(t) + \alpha\delta^h_j x_i + \beta\Delta w^h_{ij}(t-1).$$

9. Repeat steps 4 to 8 until the *total error* of the network $E = \frac{1}{2}\sum_{i=1}^{r}(y_i - o_i)^2$ is small enough for each of the training-vector pairs in the training-set.

It needs to be said that this algorithm does not correspond to the process of learning in the actual brain. The resulting network after training, however, is assumed to employ some of its basic computational principles. There are other network learning rules than backpropagation that more closely correspond to the actual learning process employed by the natural brain.

A simple example: exclusive OR

We will now give an example of a small problem: learning the logic exclusive OR function. This problem is of some historical interest because in "Perceptrons" Marvin Minsky and Seymour Papert [MINS69] showed that it was not possible to make a network for this problem without a hidden layer (for which no learning rule was known at the time). The trivial proof of this was generalized and expanded by them. This was enough reason for most researchers at the time to stop working on neural networks, because the book left the impression that neural networks had been *proven* to be a dead end.

The logical XOR function is a function of two binary variables:

`f(0,0) = 0`,
`f(0,1) = 1`,
`f(1,0) = 0` and
`f(1,1) = 0`.

A possible network configuration for this problem is shown in figure 5. This network was trained until convergence had occurred. The numbers next to the connections in the figure correspond to the weights, the numbers at the nodes with the biases. The `(I,O)` pairs were randomly selected from the four different possibilities with the booleans 0 and 1 represented by 0.1 and 0.9 respectively. These values 0.1 and 0.9 are not really important for the inputvector, because the weights and the biases are able to scale and translate it. But for the outputvector (here just one number) it is important, since the sigmoid function used to calculate the activation of the output nodes keeps the activations within the interval (0,1). Output nodes and hidden nodes, can never have an activation of exactly 0 or 1, for that would require an infinite negative or positive input. Learning the XOR function with this network takes a lot of training, about 2000 input/output pairs need to be presented, and sometimes it will not learn the

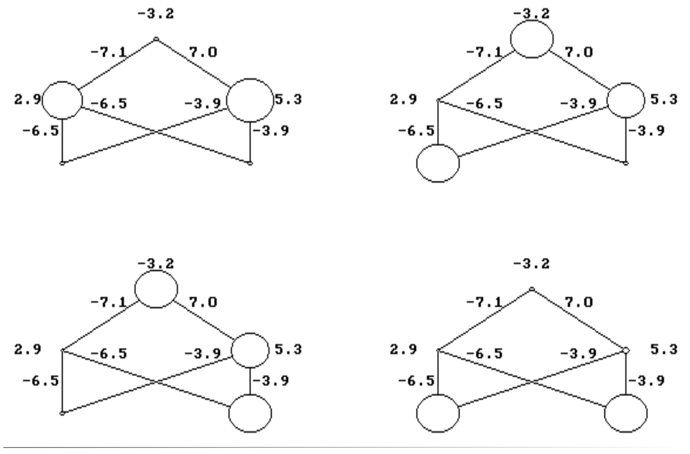


***Figure 5.** This figure shows a solution for the XOR problem. The radius of the circle is proportional to the activation of the node.*

problem at all because it can get stuck in a local optimum. The probability of this to happen depends on the learning parameters and the range of the random initializations of the weights.

Selecting the parameters for backpropagation

In the last paragraph we suggested some values: [-1,1] for the initial random weights, [0.1,0.5] for the learning-rate and [0.8,0.95] for the momentum parameter. The optimal settings of these parameters might, however, strongly depend on the task that has to be learned.

To get a feeling for the complexity of backpropagation, imagine a landscape with hills and valleys. The position of a point on the surface corresponds to the weights of a network. The height of that point depends on the total error of that network, given the coordinates (weights) of that point. Generally, it is impossible to show how such a *weight-space* looks in reality because we are unable to see more than 3 dimensions (our simple XOR network already has 9 dimensions, since each weight and bias corresponds to one dimension). But it is possible to take a two-dimensional hyperplane through the weight-space.

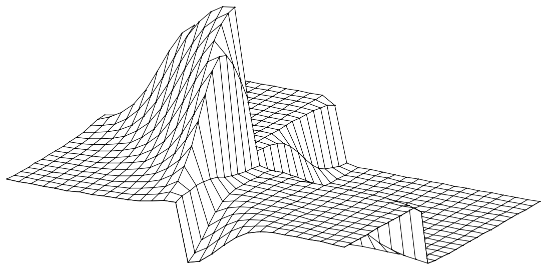


Figure 6. *A hyperplane of the 9 dimensional weight-space of the XOR example network.*

Figure 6 gives an example of such a surface of our XOR network. The x-axis corresponds to the weight from the first input node to the first hidden node and the y-axis corresponds to the weight from the second input node to the first hidden node. All the other weights are taken from our example solution from figure 5, and are not changed. This is just one of the possible $\binom{9}{2} = 36$ combinations of two out of nine weights. The figure is created by varying the two weights along the axes (the picture is made with a range [-44,44] for both axes) and measuring the performance (the total error) of the resulting network. The figure is drawn upside-down, so the best weight-configuration corresponds to the highest point on the drawing. Note that this drawing is just one of many possible two-dimensional hyperplanes in the real weight-space of this particular XOR-network. If the same network would have been trained using other initial weights the drawing might have looked quite different.

Looking at figure 6 it is clear why it often takes such a long time before a network finds a good solution. As explained in appendix A, learning in backpropagation is a gradient descent process. The next change of the network, ignoring the momentum for a moment, equals the negative gradient of the position on the weight-space-surface multiplied by α. It can be compared with someone standing on a hill, always taking a step in the downhill direction, where the size of the step depends on the slope of the surface at the position where the person stands. If this slope is very small, the steps, as a consequence, will be very small. This shows why there is a need for a momentum. The momentum tends to add all these small steps together, so small steps all in the same direction will gradually increase the momentum term and thereby increase the speed of convergence.

Because the sigmoid function used in the backpropagation algorithm has its steepest slope around the origin and the steps taken by the learning algorithm are proportional to the derivative of this sigmoid, it is usually best to take small initial weights in order to start the process of learning at a good place. This can also be seen in figure 6, where the steepest slope is around the origin. In figure 4 it can be seen that a large slope is present if the total stimulation of a node is somewhere between -3 and 3. This suggests a better way to determine in advance the range from which the random initialization should be chosen. Without loss of generality, we suppose all inputvector-values are in the range (0,1) because this corresponds to the activation range of all nodes not in the input layer. Now suppose we have a node with input from just one other node. This input is between 0 and 1. To prevent the input of that node to be in an area of the sigmoid function with a small slope, we have to give the weight of that input a value between -3 and 3. If a node has more than one input, say *n*, we have to make sure the total stimulation stays between -3 and 3, so the random weights should be chosen from a smaller range. If we simply divide this range of the random weights by the number of inputs, *n*, we are sure the total stimulation will stay between -3 and 3.

But on average, because the standard deviation of the sum of *n* numbers divided by *n* will be $\sqrt{n}$ times as small as the standard deviation of each individual number, the range of the total stimulation will on average be $\sqrt{n}$ times too small. So we propose to take the initial weights of each node to be in the range $\left[\frac{-3}{\sqrt{n}}, \frac{3}{\sqrt{n}}\right]$. If this range is used to calculate the random initial weights, the network will always have a reasonable initial weight setting no matter what the size of the network may be.

The best choice of the learning-rate parameter α (α > 0) and the momentum parameter β (0 ≤ β < 1) is also very dependent on the size of the network and the problem at hand, and usually the optimal setting is unknown. An optimal setting for α and β should result in a fast and correct convergence of the network. To get an impression of the influence of α and β on the learning process, take a look at figure 7. It is a hypothetical 1-dimensional cross-section of an error surface like figure 6 (but not upside down). Suppose we do not use the momentum (β=0). The ball representing the current state of the network, positioned at A, rolls down to B, but will stay there because it does not acquire any momentum. If we *do* use momentum, but not enough, the ball will roll to just before C, then roll back and start a damped oscillation around the local minimum B. If the momentum term is large enough, the ball will *overshoot* C and will fall in the global minimum D. If the momentum is too large, the ball may roll up towards E and back again over C into B. With a momentum parameter setting of 1, the ball will keep moving forever. This can be compared with moving without friction. The setting of the learning-rate parameter α should not be too large, to prevent missing the wanted minimum in the error-surface by taking too large steps. However, a very small setting of α will drastically increase the learning time.

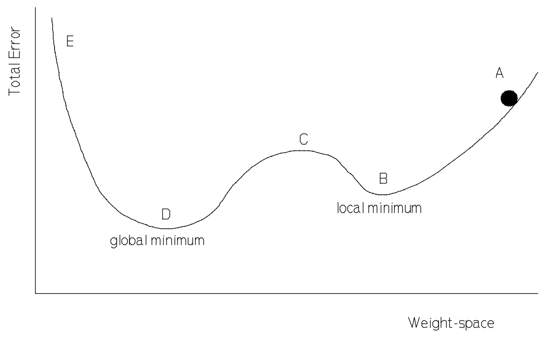


Figure 7. *Imaginary weight-space.*

Figure 8 gives an impression of the influence of the settings of α and β for the XOR example of figure 5, varying both parameters from 0.05 to 0.95. It shows the average number of iterations needed to converge as a function of α and β. The least iterations were needed at α=0.6 and β=0.65.

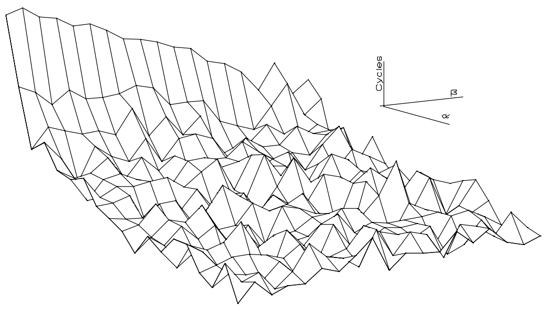


***Figure 8.** Average amount of training cycles needed to train the network of figure 5 as a function of α and β.*

Problems with backpropagation

As mentioned in the last paragraph, it is possible for backpropagation to get stuck in a local minimum. When this local minimum performs only slightly worse than the global minimum it may not be a problem, but usually a local minimum is not a good solution. As shown, the momentum term is able to overcome this problem in some cases.

One of the reasons why a backpropagation network sometimes does and sometimes does not converge is the random initialization of the weights. For some initial weight settings, backpropagation may not be able to reach the global minimum of the weight-space, while for other initializations the same network *is* able to reach it. And even when during training the network reaches the global minimum, the network may not be useful. Suppose, for example, we want to make a network able to recognize handwritten digits. It is clearly impossible to train the network with *all* possible handwritings, so a small set of *examples* has to be made and presented to the network. This set of examples may be perfectly learned by the network, but this does not say anything about how the network will respond to other handwritings. This property of being able to respond correctly to input *not* seen during training is called *generalization*. Generalization can be compared with *interpolation* in mathematics. Backpropagation usually generalizes quite well, but sometimes it does not. One of the reasons why it may not generalize is an effect called *overtraining*. This happens when a small set of examples of the total task domain is trained for a very long time. The network initially learns to detect global features of the input, and as a consequence generalizes quite well. But after prolonged training the network will start to recognize each individual input/output pair rather than settling for weights that generally describe the mapping for all cases. When that happens the network will give exact answers for the training set, but is no longer able to respond correctly for input not contained in the training set. In that case it is better to stop training before the network has converged. Overtraining is most likely to happen in very large networks, because they can easily memorize each individual input/output pair. The usual way to overcome this problem is to train a smaller network: it can not learn the training set as good as before, but it will be able to respond correctly for other input vectors. Another method is to add noise to the input data. This makes sure each input/output pair is different, preventing the memorization of the complete training set.

Unfortunately backpropagation is not suitable for *extrapolation*: it cannot give good answers for input vectors that are outside the domain of the training set. If for example a network is trained to give the cosine of its input for values between 0.5 and 0.9, it will not be able to give correct answers outside that range.

One last problem to be mentioned here is the problem of *interference*. Interference occurs if a network is supposed to learn several unrelated problems at the same time. With a very small network it may be that the network is simply not able to learn more than one of the problems. But if the network is in principle large enough to learn all the problems at the same time, this may still not happen. The different problems seem to be in each others way: if one of the problems is represented in the weights of the network the other problem is forgotten, and vice versa. An example of such interference between more classifications is the recognition of both position and shape of an input pattern (see [RUEC89]). Rueckle et al. conducted a number of simulations in which they trained a three layer backpropagation 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. So there were 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. Of importance was the number of hidden nodes allocated to both sub-networks. 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* network, and 14 to the apparently more complex task of the *shape* network. It needs to 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 un-split hidden layer caused interference. What they failed to describe is that removing the hidden layer completely, connecting input and output layer directly, leads to an even better network than the optimum they found using 20 hidden nodes in separate sub-networks.

Analysis of these results revealed that the processing of what and where strongly interfered in the non-split model. The non-split model was not constrained to develop in a particular way, and in principle could have developed the same configuration of weights as the split network (no connection being represented with a zero weight), but the chance of this to happen is very small.

The problems mentioned above are not restricted to backpropagation. Most other neural network paradigms suffer from the same problems. There seems to be a solution for these problems, already mentioned in the introduction of the brain, and hinted at in the last two sections: *modularity*. In chapter 5, we will return to this, but here we will restrict ourselves to showing how modularity can be incorporated in backpropagation.

Modular backpropagation

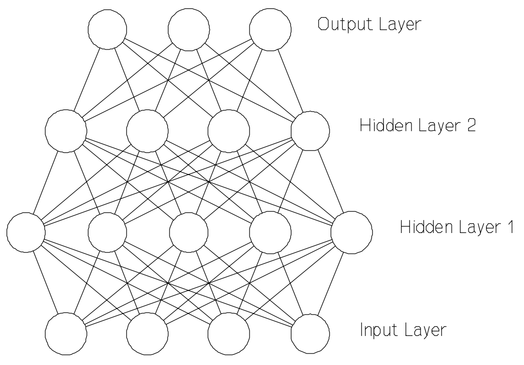


Figure 9. *A network with two hidden layers.*

Until now we have only discussed simple backpropagation network topologies consisting of an input layer, a hidden layer and an output layer. The learning algorithm can be used for other types of network topologies as well, as long as the networks have a feedforward structure. One of the most simple changes in structure is the addition of an extra hidden layer, like in figure 9. A network with just one hidden layer can compute any function that a network with 2, or even more, hidden layers can compute: with an exponential number of hidden nodes, one node could be assigned to every possible input pattern (see e.g. [HECH90]). However, learning is sometimes much faster with multiple hidden layers, especially if the input is highly nonlinear, in other words, hard to separate with a series of straight lines. The learning algorithm does not have to be modified, all hidden layers can simply take the errors calculated at the next layer, whether output or hidden, to calculate their own errors.

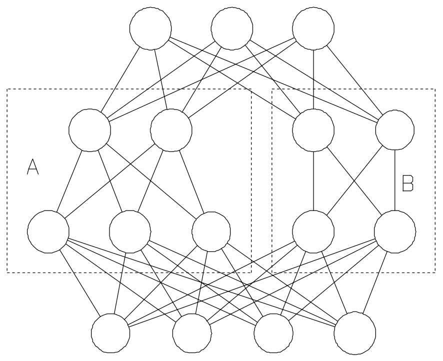


Figure 10. *An example network made of two separated hidden sub-networks.*

Because with more hidden layers each node in a layer is still connected to the same nodes as all the other nodes in that layer, only the weights are able to make each node in a layer behave differently to the rest. Another way to accomplish this is to make different nodes see different things, that is, connect the nodes in one layer differently. Instead of full connectivity between two layers, specific connections can be left out.

So the idea of adding more hidden layers can be greatly extended with the possibility of splitting up several layers into sub-layers, hereby reducing the number of weights. Take for example figure 10 which shows an example of a division of the network from figure 9 into two separate parts A and B. Because there are no connections between both parts, the dimensionality (the number of weights) of the weight-space is reduced by 10. This not only decreases the amount of computing, but also may take away several local minima and increase the speed of convergence. It can now be seen that besides a vertical organization in layers also a horizontal organization is possible.

In order to work more flexible with all kinds of different network topologies we define a *module* to be a group of unconnected nodes, each connected to the same set of nodes. So the network of figure 9 is made of 4 modules and the network of figure 10 is made of 6 modules. The set of weights between two modules are also grouped together to form a *connection*. There is no loss of generality here, because all possible feedforward network structures can be built with these components: if necessary one can make modules consisting of just 1 node. The network of figure 10 can now more easily be visualized as in figure 11.

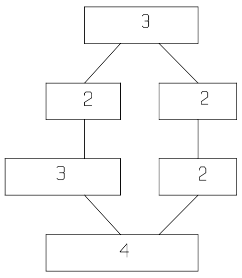


Figure 11. *A modular network.*

As an example of how much depends on the structure of the network, remember the XOR network shown in figure 5. The figure showed one possible solution, but on many trials, the network did not converge. Rumelhart and McClelland [RUME86] describe several occasions where the network of figure 5 got stuck in a local minimum. By changing the network topology we found a network (figure 12) that, for as many times as we have tested it, always learned the XOR problem. Not only did it *always* learn the problem, but it did it much *faster* than the simple network in figure 5. We tested both networks the same way, by training the network a 100 times, for all possible combinations of the parameters α and β (from 0.05 to 0.95 with steps of 0.05). The results of this simulation for the original network are shown in figure 8 and for the network of figure 12, which has 2 additional weights from the input nodes directly to the output node, in figure 13. The original network

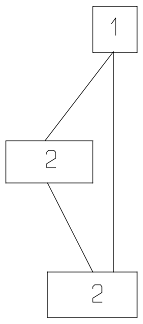


Figure 12. *XOR network.*

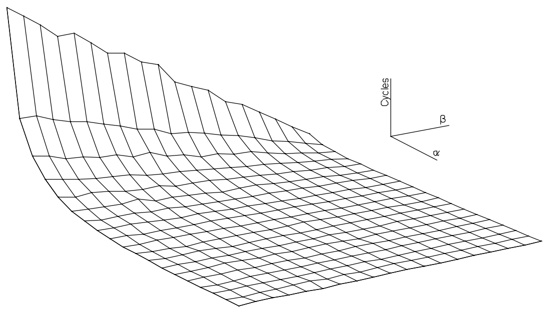


Figure 13. *Number of training cycles needed for XOR by figure 12 versus α and β.*

needed on average 1650 training cycles for the best combination of α and β, while the new network needs only thirty training cycles on average. The large fluctuations in figure 8 signify the dependence on the initial weights of the original network, and conversely figure 13 shows the independence of the initial weights of the new network.

The intuitive idea behind making different network topologies for different problems can be explained as follows. It may be seen as moulding the weight-space in such a way that all local minima disappear, and no matter from where you start the training, there always is a clear path from that starting point to a global minimum in weight-space.

If a network with one hidden layer with *N* nodes finds a solution, a permutation of the nodes in the hidden layer, together with all their incoming and outgoing weights, will of course result in the same solution. This means that when the training process starts, and there is a solution, at least as many as *N*! possible solutions have to exist. If two or more nodes have the same input and output weights, which is very unlikely, some permutations will result in the same network. More likely, there will be several different solutions, with respect to all possible permutations. This was frequently observed when testing the 'standard' XOR example. For networks with more hidden layers the number of possible permutations of the subsequent layers may be multiplied, so the total number of permutations possible with a network with *i* hidden layers with N_i nodes each (ignoring similar nodes, and different solutions with respect to permutation) is given by:

$$\prod_i N_i! .$$

This gives a strong indication of *why* modular networks may be better at generalization than fully connected networks. The network from figure 9 for example has 4!5! = 2880 permutations, while the network shown in figure 10 has only 3!2!+2!2! = 16 permutations, which largely reduces the amount of ambiguity of the eventual solution after training. This notion of the number of possible solutions is generalized in appendix B.

Now a major problem arises: given a particular problem, how to find a network topology that is optimal for this problem, which means:

- always converge, independent of arbitrary settings of α, β and initial weights,
- converge as fast as possible,
- show no interference,
- be able to generalize.

At present there is little theory on how to design a good network topology for a particular problem. Usually people working with backpropagation use some rules of thumb, like:

- take the number of hidden nodes equal to the average of the number of nodes in the input and output layer,
- if the network does not converge take more hidden nodes,
- if that does not work add a hidden layer,
- if the network does not generalize well, take less hidden nodes,

and so on. There is not yet any method available in order to find a good (modular) network. Finding such structured networks was the main subject of this research. In chapter 5 we will return to this problem, and offer a possible solution.

Implementation

The modular backpropagation network algorithm used in this research has been implemented in C++, and runs under MS-DOS as well as UNIX. It allows a very flexible use of modular networks. The library can easily be linked to other programs, and has already been used for demonstration software, games and all simulations throughout this research. Plans are made to extend the program to other network paradigms, the object oriented approach should then make it possible to make networks with several different kinds of network paradigms combined.

'All modern men are descended from a worm-like creature, but it shows more on some people.'
— Will Cuppy

3 Genetic Algorithms

One of the problems mentioned in the previous chapter is the problem of finding a topology able to learn a specific task. It seems appropriate to use a computer program to relieve us of this difficult, time consuming task. Traditional search methods however, are not suitable for the task because of the vast number of possible connections between the nodes and because little is known about why one topology is better than another. Recently, search methods have been developed which can handle such multiple constraint problems. One of them, *genetic algorithms*, will be treated in this chapter.

Genetic algorithms, introduced by John Holland [HOLL75], are based on the *biological metaphor of evolution*. In his recent book, David Goldberg [GOLD89] (p.1) describes genetic algorithms as '... *search algorithms based on the mechanics of natural selection and natural genetics* [resulting in] *a search algorithm with some of the innovative flair of human search*.'

Overview

Goldberg mentions the following differences between GAs (genetic algorithms) 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 a problem are coded into a string of (usually) binary features (analogous with *chromosomes* in biology). This coding is done by the user of the GA. The GA itself has no knowledge at all of the meaning of the coded string. If the problem has more than one parameter, the string contains multiple sub-strings (or *genes*), one for each of the parameters. Each coded string represents a possible solution to the problem. The GA works by manipulating a population of such possible coded solutions in a *reproduction process* driven by a number of *genetic operators*.

	Bitstring:	Fitness:
1	1100010101	9
2	0000100010	7
3	1000000001	6
4	0001100010	5
5	1101110101	5
6	0001000100	4
7	1111111000	3
8	0000000001	3
9	1100001000	2
10	1111111111	1

Figure 1. *Sample population of 10 strings. (The population is sorted, although that is not necessary for roulette wheel selection).*

During the reproduction process, new solutions are created by selecting and recombining existing solutions based on pay-off information (often called the *fitness*) using the genetic operators. The process can be compared with natural selection and the Darwinian theory of evolution in biology: fit organisms are more likely to stay alive and reproduce than non-fit organisms.

The fitness of a string (or solution) can be evaluated in many different ways. If the problem, for example, is finding the root of a mathematical function, the fitness can be the inverse of the square of the function value of the proposed solution. If the problem is finding an optimal neural net, the fitness could be the inverse of the convergence time and zero if the network couldn't learn the problem. It could also be the inverse of the error at the output nodes. The GA is not aware of the *meaning* of the fitness value, just the value itself. This implies that the GA can't use any auxiliary knowledge about the problem. Figure 1 shows a sample population of 10 strings, and represents all the knowledge a GA would have.

Starting with a population of random strings, each new population (generated by means of reproduction) is based upon (and replaces) the previous generation. This should, in time, lead to a higher overall fitness, and thus to better solutions to the original problem.

The four most commonly used genetic operators used are *selection*, *crossover*, *inversion* and *mutation*. With each of these operators, only random number generating, string copying and changing of bits are involved. Crossover, mutation and inversion are all applied with a certain probability: for each application of an operator it must be decided whether to apply the operator or not. Selection alone is usually not enough for the GA to work, so, one or more of the other genetic operators have to be applied to the selected string(s).
In what follows, an explanation of the genetic operators is given. After that, a more thorough treatment is provided of the basic mechanisms of genetic search as implemented by a GA.

Selection

Selection is used to choose strings from the population for reproduction. In parallel with the natural selection mechanism, strings (solutions) with a high fitness are more likely to be selected than less fit strings. The two selection methods applied in this research are described respectively by Goldberg [GOLD89] and Whitley [WHIT89A].

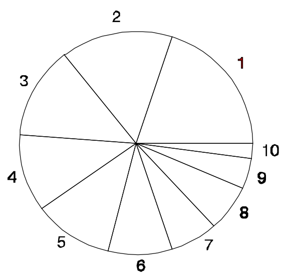


Figure 2. *The roulette wheel for the sample population from figure 1. Member 1 has a 20% chance (9/45) of being selected.*

With *roulette wheel selection* [GOLD89], strings are selected with a probability proportional to their fitness.

Another method is called *rank based selection* [WHIT89A], where the chance of being selected is defined as a linear function of the rank of an individual in the population. The population must remain sorted by fitness for this method to work. One advantage of rank based selection is that it does not need the fitness scaling necessary with other methods, to prevent high fitness strings from dominating the population, which may result in a premature convergence into a non-optimal solution.

Crossover

The crossover operator creates new members for the population by combining different parts from two selected parent strings.

First, a number of *crossover points* (usually two) are chosen at random. A new string is created by using alternate parts of the parent strings. A sample crossover with two crossover points is shown in figure 3.

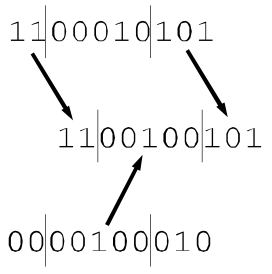


Figure 3. *Crossover of members 1 and 2.*

Inversion

Inversion is an operator that reorders the positions of genes within the chromosome (string). Two inversion points are chosen at random between genes, and the genes between the two points swap places: the first is swapped with the last, the second with the last but one, etc.
A new member is constructed by concatenating the reordered genes together.

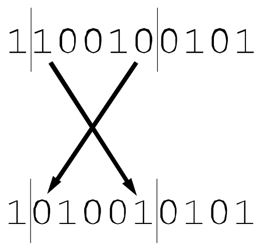


Figure 4. *Inversion of the crossover result from figure 3.*

An example is shown in figure 4, using strings with genes containing only 1 bit.

Inversion complicates the GA somewhat because one has to keep track of the position of each gene within the chromosome. To prevent mixing different parameters during crossover (because two strings have a different gen order), the genes are put temporarily in the same order before crossover is applied. The newly created string inherits the gen order from one of the two parents.

Mutation

Mutation is possibly the simplest of the genetic operators. It randomly flips bits in the string from 0 to 1 or from 1 to 0.

The purpose of this string mutation is to improve the algorithm by introducing new solutions not present in the population, and by protecting the algorithm against accidental, irrecoverable loss of (valuable) information due for example, to unfortunate crossovers.

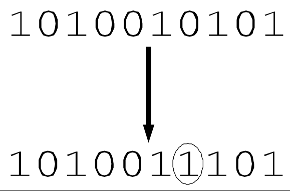


Figure 5. *Mutation of the inverted string. Bit 7 has been mutated.*

In order to keep the algorithm from becoming a simple *random search*, mutation rate has to be low, so it doesn't interfere too much with crossover and inversion. There are some applications however, where selection and mutation are enough for the GA to function (e.g. [GARI90]).

Building blocks

So far, it may not be clear how and why those simple genetic operators combine into such a powerful and robust search method. Or, as Goldberg [GOLD89] (p.28) describes it: '*The operation of genetic algorithms is remarkably straightforward. After all, we start with a random population of strings, copy strings with some bias toward the best, mate and partially swap sub-strings, and mutate an occasional bit value for good measure.*'

The principle of structured information exchange with crossover and inversion leads to an exponential increase of high fit pieces of genetic information (partial solutions) which are combined to produce good overall solutions. To explain this, let us first take a look at an example.

Bitstring:	Fitness:
00001101	169
11000110	39204
00111001	3249
11110001	57600

Figure 6. *Example with $F = x^2$. Four members with their fitness.*

If we want to maximize the function $F(x) = x^2$, for x in the range 0..255, we can take binary strings (chromosomes) of length 8, with x coded as the binary representation of x. The best solution is of course $x = 255$, because x^2 is maximal for the largest possible x. But if we have the strings shown in figure 6 in our population and we don't know anything about the fitness function F, the best way to proceed would be to look for *similarities* between highly fit strings if we wanted to create a new, more fit, string. In figure 6, it appears that having 1's on the first positions results in a high fitness. Therefore it might be a good idea to put those 1's in the first positions of the new string too. This idea of using similar (small) parts of highly fit strings to create a new string can be explained more precisely using the concepts of *schemata* and *building blocks*.

A *schema* (introduced by John Holland, [HOLL68] and [HOLL75]) is a template describing a subset of strings with similarities at certain string positions. If we take for example a population of binary strings, schemata for these strings are strings themselves, consisting of 0, 1 and * symbols. The * (*wild card* or *don't care* symbol), matches either a 0 or a 1. A schema *matches* a particular string if at every position a 1 in the schema matches a 1 in the string and a 0 matches a 0 in the string. If we take strings of length 8 (binary representations of the numbers 0..255) the schema 1*0001*0 matches four strings: 10000100, 10000110, 11000100 and 11000110.

First, let us take a look at the number of schemata involved in a population of n strings of length l. If each string is built from k symbols (k=2 for binary strings), there are $(k+1)^l$ different schemata (because each of the l positions can be one of the k symbols, or an asterisk). So, for our example, there are only 256 (2^8) different strings, but there are $(2+1)^8 = 6561$ different schemata. Also, a string of length 8 belongs to 2^8 different schemata because each position may take on its actual value or a wild card (*) symbol. For strings of length l, this number is 2^l. So, for a population of size n, the population contains somewhere between 2^l and $n \cdot 2^l$ schemata. So even moderately sized populations contain a lot of information about important similarities. By using schemata in the genetic search, the amount of information can be much larger than by looking at the strings only.

Every schema can be assigned a fitness: this is the average fitness of the members in the population corresponding to that particular schema. We will denote this average fitness with f_s. Also, every schema has a *defining length* (or δ) which is the distance between the first and the last *non*-wild card. Looking at the defining length, we note that crossover has a tendency to cut

schemata of long defining length when the crossover points are chosen uniformly at random: for example, the schema 1*****10 has a higher chance of being cut than *****10* (6/7 or 86% vs 1/7 or 14%).
A lower bound on the *crossover survival probability* p_s for a schema with defining length δ can be expressed with the following formula (for crossover with one crossover point):

$$p_s \geq 1 - p_c \frac{\delta}{l-1} \tag{1}$$

where p_c is the probability with which crossover will occur, δ is the defining length of the schema, and l is the length of the schema. The formula contains a '≥' instead of a '=' because even when the schema is cut it can survive if the crossover results in a string that still contains the schema. New strings with the schema can also come into existence.

We can also calculate the effect of *selection* on the number of schemata. When we have *m(t)* examples of a particular schema at time *t* in our population, we can expect

$$m(t+1) = m(t)\, n \frac{f_s}{\sum_{i=1}^{n} f_i}$$

examples at time *t*+1, where *n* is the population size, f_s the average fitness of the strings representing the schema and $\sum_{i=1}^{n} f_i$ the total fitness of the population. If we rewrite the formula, using

$$f_{avg} = \frac{1}{n} \sum_{i=1}^{n} f_i$$

for the average fitness of the whole population, it becomes:

$$m(t+1) = m(t) \frac{f_s}{f_{avg}} \tag{2}$$

Or: a particular schema grows as the ratio of the average fitness of the schema and the average fitness of the population. So schemata with fitness values above the average population fitness have a higher chance of being reproduced and receive an increasing (exponential) number of samples in the new population. This is carried out for *each* schema in the population in parallel.

Because mutation has only a very small effect on the number of schemata (mutation rate is usually chosen very low), the combined effect of selection and crossover can be expressed with the following formula, which is the result of combining (1) and (2):

$$m(t+1) \geq m(t) \frac{f_s}{f_{avg}} \left[1 - p_c \frac{\delta}{l-1} \right] \tag{3}$$

So a particular schema grows or decays depending upon a multiplication factor. With both selection and crossover the factor depends upon whether the schema's fitness is above or below the population's average fitness and on the length of the schema. Especially schemata with high fitness and a short defining length are propagated exponentially throughout the population (the *Schema Theorem*, [GOLD89]). Those short schemata are called *building blocks*. Crossover directs the genetic search towards finding building blocks (or partial solutions) and also combines them into better overall solutions (the *building block hypothesis*, [GOLD89]). Inversion also facilitates the formation of building blocks. Complex problems often consist of multiple parameters which are coded by different genes on the chromosome. With these multiple parameter problems however, complex relations may exist between different parameters. When defining the coding of such a problem, related genes should be positioned close together. When not much is known about the relations between the parameters, inversion can be used as an automatic reordering operator.

Implicit parallelism

The exponential propagation of high fit, small size schemata (building blocks) goes on in parallel, without any more special bookkeeping or memory than a population of *n* strings. Goldberg [GOLD89] presents a more precise count of how many schemata are processed usefully in each generation: the number turns out to be roughly n^3. Because only *n* function evaluations (fitness calculations) are done for each generation, this feature has also been called *implicit parallelism,* and is apparently unique to genetic algorithms.

Applications

The genetic algorithms described, and many variations, are still a active topic of research. However, they are used in many applications already, and in this paragraph a few are mentioned. Also one application is described in more detail as an example.

Goldberg's book [GOLD89] contains a table with an overview of the history of genetic algorithms. Below is an extract from that table, which shows the diversity of problems where genetic algorithms have been applied. The table shown here is far from complete, and new applications are found continuously.

Year	Investigators	Description
	Biology	
1967	Rosenberg	Simulation of the evolution of single-celled organism populations
1987	Sannier and Goodman	GA adapts structures responding to spatial and temporal food availability
	Computer Science	
1979	Raghavan and Birchard	GA-based clustering algorithm
1985	Rendell	GA search for game evaluation function

Year	Investigators	Description
	Engineering and OR	
1983	Goldberg	Optimization of pipeline systems
1985	Davis and Smith	VLSI circuit lay out via GA
1986	Goldberg and Smith	Blind knapsack problem with simple GA
1986	Minga	Aircraft landing strut weight optimization
	Miscellaneous	
1985	Brady	Travelling salesman problem via genetic-like operators
1985	Gillies	Search for image feature detectors via GA
1981	Smith and De Jong	Calibration of population migration model using GA search

With the travelling salesman problem (TSP) mentioned in the last row, a hypothetical salesman must make a complete tour of a given number of cities in the order that minimizes the total distance travelled. He should return to the starting point and no city may be visited more than once. Although it may seem a trivial problem, it is NP-complete which means it is currently not solvable in deterministic polynomial time. Because of this, research has been done using GAs in order to find (near-)optimal solutions in reasonable time.
Experimenting with GAs, we also have tried to solve the TSP problem (using only a variation of the inversion operator) and it did find (near-)optimal solutions in time roughly proportional to N^2 (with N the number of cities). As described before, the GA had no knowledge at all of the problem (so no graph theory could be used), only the fitness of each member. The used variation on inversion (swapping of a sub-path in the whole path) propagates the earlier described building blocks through the population. Figures 7 through 10 show some of the best members of subsequent populations found during a simulation with thirty cities. Figure 7 is the starting best member, figure 10 is the found optimum. The total travelling length for each solution is also given. In figure 11, the fitness of the best solution (the length of the path) is given against the number of recombinations so far. The GA converged quickly to a length of around 3000 and then slowly to the found optimum.

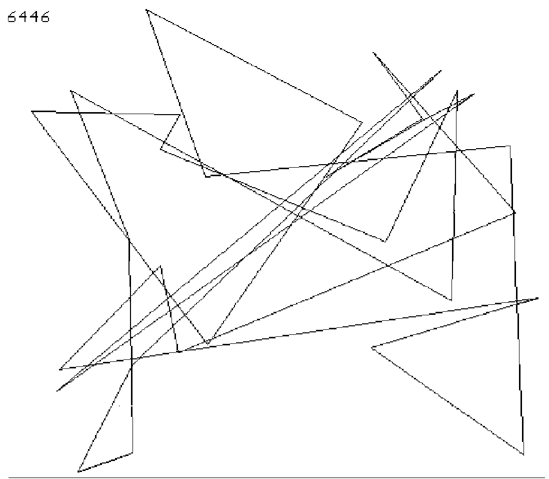


Figure 7. The path from the starting solution.

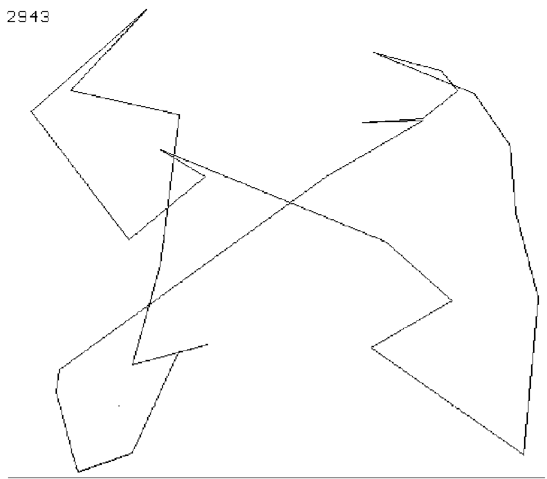


Figure 8. The GA quickly converged to this solution.

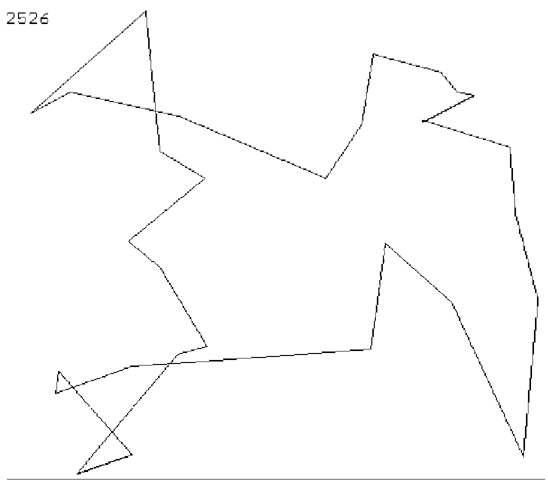


Figure 9. *The path after 50500 recombinations.*

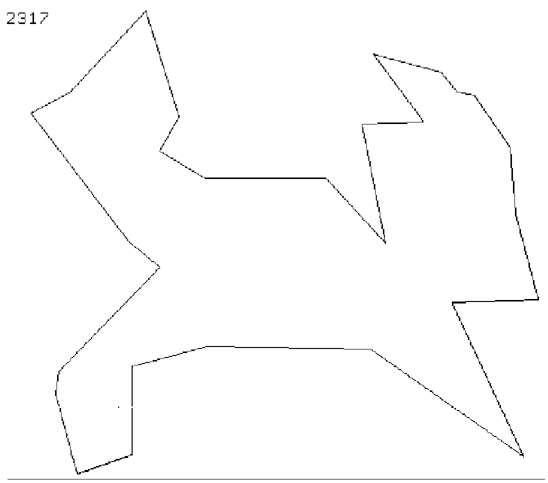


Figure 10. *The optimum solution found.*

Implementation

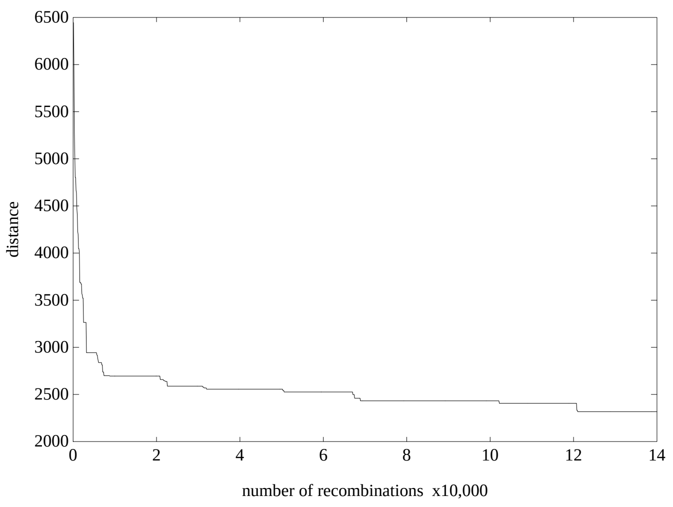


Figure 11. *Length of best member against number of recombinations.*

Goldberg's book contains programs to implement a genetic algorithm (in Pascal). Some changes to the standard algorithm proposed by Whitley are described in [WHIT89A]. His program, called GENITOR, besides using rank based selection, also uses *one-at-a-time selection and replacement*: a new solution replaces the worst member of the population when the new solution has a higher fitness. The best solutions always stay within the population and therefore the best fitness value increases monotonously (the *static population model*). One-at-a-time replacement is always better than creating a whole new population if it is possible to evolve from a string with good fitness to a string with a fitness that is a global maximum without passing any local maxima. Then it will be assured that the best string in the population does not lead to a local maximum.

At the department of Experimental and Theoretical Psychology at the Leiden University, where this research took place, a library was written (in C) to create and manipulate populations of binary strings either using Goldberg or Whitley selection and replacement ([HAPP92] and [MURR92]). To make the functions more convenient for our purposes, several changes and improvements have been introduced to the original library (see also chapter 7).

‘*The ignorant man marvels at the exceptional; the wise man marvels at the common; the greatest wonder of all is the regularity of nature*.’
— George Dana Boardman

4 L-systems

The third main ‘real world model’ used in this research is, again, based upon a biological example: the development and growth of living organisms.

Biological development

The development of living organisms is governed by genes. Each living cell contains genetic information (the *genotype*) which determines the way in which the final form of the organism will develop (the *phenotype*). This genetic information is not a blueprint of that final form, but can be seen as a recipe. For example, in his excellent book "The blind watchmaker", Richard Dawkins [DAWK86] describes how this ‘recipe’ is followed not by the organism as a whole, but by each cell individually. The shape and behaviour of a cell depend on the genes from which information is extracted. This in turn depends upon which genes have been read in the past and on influences from the environment of all the neighbouring cells. Therefore the development is solely governed by the local interactions between elements that obey the same global rules. Such a principle lies also at the basis of a mathematical system called *fractals*.

Fractals

Fractals have been made popular by Benoit Mandelbrot with his book "The fractal geometry of nature" [MAND82]. Before describing one of the oldest examples of a fractal, the Koch-graph, let’s take a look at an experiment by meteorologist Lewis Richardson. He tried to measure the length of the perimeter of the west-coast of England and found that the results depended strongly upon the scale of the map he used. Repeating the experiment using just one map with all the details on it, but decreasing the unit of measure each time, we would find that for each decrease, the length of the coast would increase.

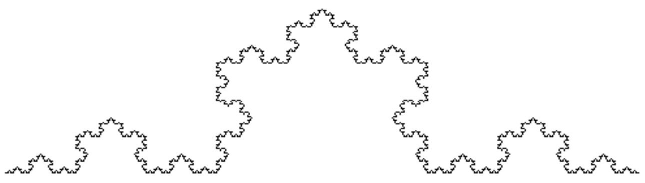


Figure 1. *One of the oldest examples of a fractal: the Koch-graph.*

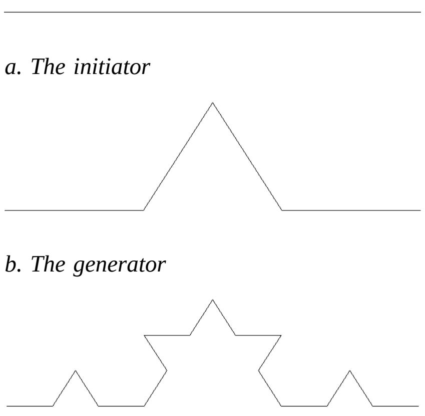
This implies that the west-coast of England has an infinite length! So it seems that the 'length' is not a very useful method of describing a coastline and that a measure of twirliness would be better. Mandelbrot called this twirliness, a number between 1 and 2, the *fractal dimension* (a fractional dimension).

If we call the unit of measure *a* and if we need to use it *N* times to make an approximation of the length of the line for which we are calculating the fractal dimension (the measured length will then be *Na*), the fractal dimension can be defined as: $D = \lim_{a \to 0} \frac{\log N}{\log \frac{1}{a}}$

The *Koch-graph* is another example of a curve with infinite length. Proposed by Helge von Koch in 1905 [KOCH05], it shocked mathematicians: a curve where each part, however small, had an infinite length! Mandelbrot restates its construction as follows:

'*One begins with two shapes, an* initiator *and a* generator. *The latter is an oriented broken line made up of* N *equal sides of length* r. *Thus each stage of construction begins with a broken line and consists in replacing each straight interval with a copy of the generator, reduced and displaced so as to have the same end points of those of the interval being displaced.*'

The resulting 'snowflake' curve (shown in figure 1) is a classical example of a graphical object defined in terms of a simple rewriting rule that is recursively applied to each element. As can be seen in figure 2c, a fractal is very *self-similar*.


a. The initiator

b. The generator

c. The generator rewritten

Figure 2. *The first steps of generating the Koch-graph.*

Simple L-systems

A special class of fractals are called L-systems and were introduced in 1968 by Aristid Lindenmayer [LIND68] 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*. 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. Whereas in other grammars production rules are applied one-by-one sequentially, in an L-system all characters in a string are rewritten in parallel to form a new string.

When we attach a specific meaning (based on a LOGO-style turtle, [SZIL79]) to the characters in a string, we are able to visualize the string.

The Koch-graph from figure 1, for example, can be described with the following L-system:

axiom F

and

production rule F $\rightarrow$ F-F++F-F.

Each of the characters can be thought of as ‘directions’ for a *turtle*. If we associate F with *draw a line*, - with *take a left turn* and + with *take a right turn*, the turtle looks at the symbols in the string, one at a time, and for each symbol, it either moves and draws a line, or it turns around. The angle for the - and + is variable and will be denoted with δ.
Using δ = 60°, figure 2a shows the axiom and figure 2b shows the right-hand-side (the part after the $\rightarrow$) of the production rule.

The left-hand-side of the production rule (the part before the $\rightarrow$) describes the sub-string to be replaced, and the right-hand-side describes the string with which it should be replaced. In the first step, the axiom will be rewritten as F-F++F-F.

In the second step, *each new* F will be replaced, in parallel, with F-F++F-F, resulting in $\underline{\text{F-F++F-F}}\text{-}\underline{\text{F-F++F-F}}\text{++}\underline{\text{F-F++F-F}}\text{-}\underline{\text{F-F++F-F}}$ (with each of the rewritten F’s underlined). If no production rule is given for a character, the character is replaced with itself. Figure 2c shows the string created from the axiom after two rewriting steps.

Besides -, + and F, more symbols can be used. For example, the f symbol tells the turtle to move, but *without drawing a line*. The difference between f and F is shown in figures 3 and 4, where δ = 90°.

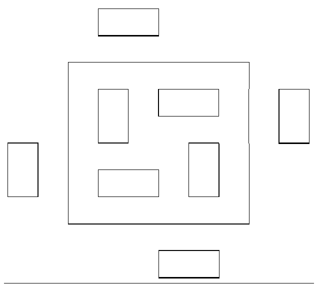


Figure 3.
Each side is represented by F-f+FF-F-FF-Ff-FF+ f-FF+F+FF+Ff+FFF.

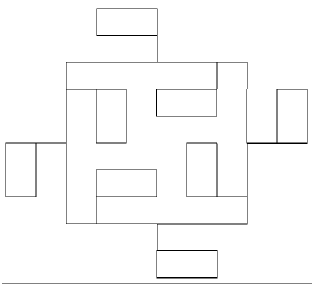


Figure 4.
Each side is represented by F-F+FF-F-FF-FF-FF+ F-FF+F+FF+FF+FFF.

Bracketed L-systems

With the turtle symbols described in the previous paragraph, only so-called *single line* drawings can be made, in contrast with the branching seen in real life plants. In order to give the turtle the ‘freedom’ to move, two new symbols are introduced:

[	Remember the current position and direction of the turtle.
]	Restore the last stored position and direction.

With these new symbols, more realistic drawings can be obtained, as shown in figures 5 and 6. Both these L-systems, and the others used in this chapter, are from the book "The Algorithmic Beauty of Plants", by Prusinkiewicz and Lindenmayer [PRUS90]. For figure 5 the X symbols are ignored during the drawing of the string.

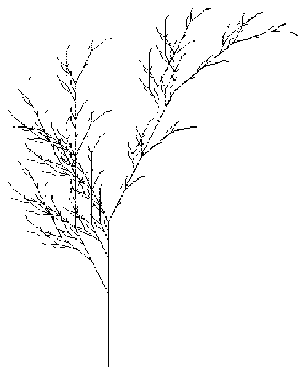


Figure 5.
5 steps, δ = 23°,
axiom X
X $\rightarrow$ F[-[[X]+X]+F[+FX]-X
F $\rightarrow$ FF

Context-sensitive L-systems

A final extension of L-systems, called *context*, is needed to model information exchange between neighbouring cells, as described in the first paragraph. Context also leads to more natural looking plants. Context can be left, right or both for a certain sub-string. A production rule in a context-sensitive L-system has the following form:

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

P (also called the *predecessor*) and S (the *successor*) are what we earlier called the left-hand-side and the right-hand-side of a production rule. L and R (the *left-* and *right-context* respectively) may be absent. Technically, an L-system without context is called a 0L-system. If all production rules have one-sided context or no context at all, it is called a 1L-system, and a 2L-system has production rules with two-sided context.

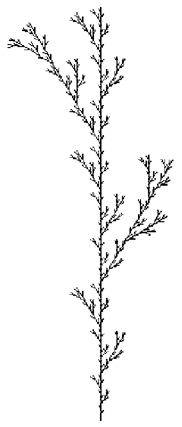


Figure 6.
5 rewriting steps
δ = 26°, axiom F
F → F[+F]F[-F]F

A production rule with left and right context L and R can only replace P by S if P is preceded by L and followed by R. If two production rules apply for a certain character, one with and one without context, the one with context is used.

For example, with the production rules from figure 7a, the string ABC will be rewritten as XYZ, but with the production rules from figure 7b, the string ABC will be rewritten as XYC. C is not rewritten because it is preceded by B *during its rewriting*, not Y.
If we try to rewrite XYZ using the rules from figure 7a the string remains XYZ (no production rules apply) but if we try to rewrite the string XYC using the rules from figure 7b it can now be rewritten to XYZ because C is now preceded by a Y.

(a)	(b)
A → X	A → X
B → Y	B → Y
C → Z	Y < C → Z

Figure 7. *Production rules with and without context.*

The plant in figure 8 was created with the production rules from figure 9. All production rules have both left and right context. Note that the turtle command F is not rewritten at all, and that only 0 < 0 > 1 and 1 < 0 > 1 actually create new twigs: all the other production rules are used for the interaction between the different parts of the plant. During context matching, the geometric symbols (-, + and F) are ignored. During the drawing of the string, the 1s and 0s are ignored. The production rules were constructed by Hogeweg and Hesper [HOGE74], along with 3583 other patterns generated by bracketed 2L-systems.

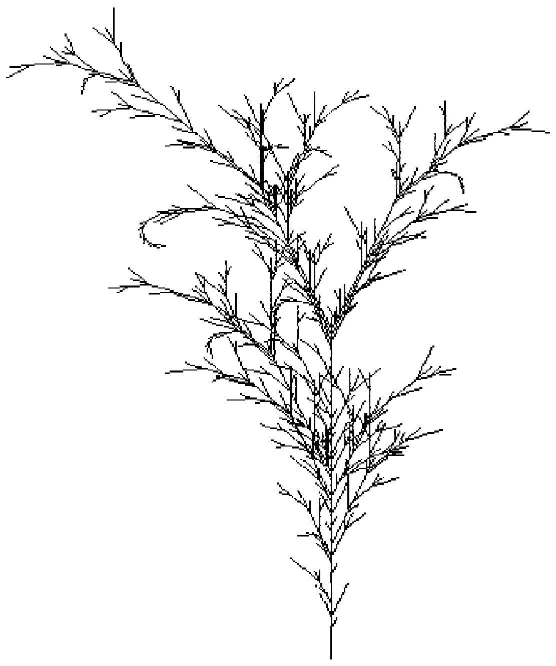


Figure 8. *30 rewriting steps*
δ = 16°, axiom F1F1F1
Production rules as in figure 8.

Implementation

Przemyslaw Prusinkiewicz and James Hanan [PRUS89] present a small L-system program for the Macintosh (in C). In order to experiment with L-systems, we have ported the source code to work on PC's. Besides fixing some 'irregularities', the program was also rewritten extensively in order to accept less rigid input files. Two features were added: *probabilistic production rules* and *production rule ranges* (both from [PRUS89]). With probabilistic production rules, more than one production rule for the same L, P and R can be given, each with a fixed probability. When rewriting a string, one of the rules is selected at random, proportional to its probability. This results in more natural looking plants, while they are still recognizable as belonging to the same ‘family’. Figure 10 shows two plants that were created with the same set of (probabilistic) production rules.

```
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
    +     → -
    -     → +
```

Figure 9. *The production rules for figure 8.*

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 example, to generate twigs first and then the leafs and flowers at the end of those twigs.

In chapter 5 a method is set forth that combines the GAs from chapter 3 and L-systems in order to obtain an efficient search for good neural network topologies.

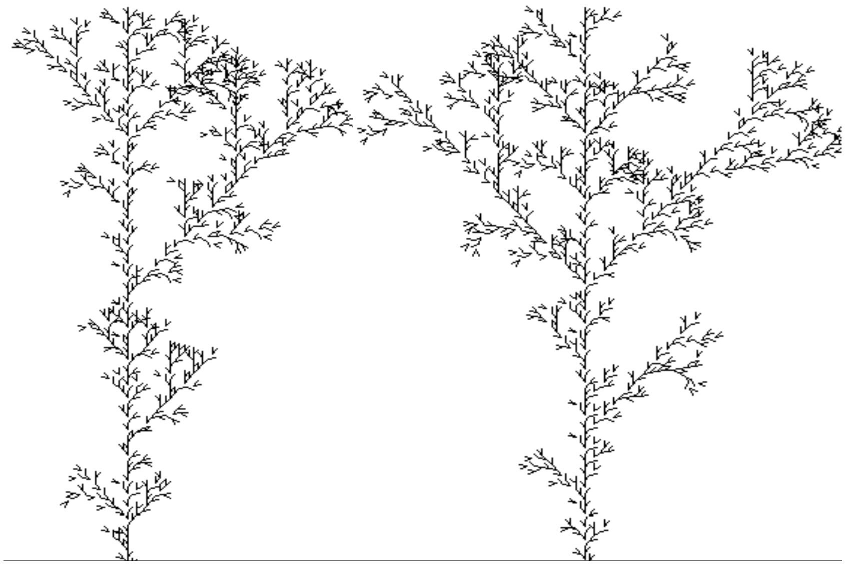


Figure 10. *Both plants were created with the rules F → F[+F]f[-F]F, F → F[+F]F and F → F[-F]F, each with a 33% probability.*

‘*Nature gets credit which should in truth be reserved for ourselves: the rose for its scent, the nightingale for its song; and the sun for its radiance. The poets are entirely mistaken. They should address their lyrics to themselves and should turn them into odes of self congratulation on the excellence of the human mind.*’
— Alfred North Whitehead

5 The search for modularity

This chapter will present an overview of the principle of *modularity* as seen in nature. It will be explained why modularity incorporated in neural networks might greatly improve their performance. It will conclude with a method to find modular structures, a method which *itself* is also based on modular principles.

Modularity in nature

Modularity is found in nature on all possible scales, in living organisms as well as in dead objects. A very broad definition of the principle of modularity may be: *a subdivision in identifiable parts, each with its own purpose or function*. This of course, applies to almost everything, but that is exactly the point to be made. Almost everything *is* modular.

At the smallest possible scale, that of quantum physics, the principle of modularity sets of. All elementary particles can be seen as the modules of which everything else is made. These particles can, however, no longer be divided into smaller particles so they themselves are at the lowest level of modularity. At a larger scale these modules (the elementary particles) form into progressively larger entities: atoms, molecules, solids and fluids, celestial bodies, star-systems, galaxies, clusters, super-clusters up to the universe itself. Between each of these scales a clear subdivision of one into the other can be made.

But not only in the domain of physics and astrophysics modularity may be found, biology is also based on modular principles. The same kind of progressive subdivision as made in the last paragraph is possible. All possible classes of living organisms are made of atoms, molecules, solids and fluids. The mammals for example, can be subdivided in their cells, each of which can be subdivided in several organelles (specialized parts of a cell) positioned in the nucleus and in the cytoplasm (the primary living matter of a cell not including the nucleus). These cells are grouped together to form organs, which in turn form the whole organism. In the organism each organ has one or more specific functions. Each organ itself is again divided into parts with different functions, and in those parts each cell has its own tasks.

These examples suggest a recursive modularity inside modularity itself. Two kinds of modularity can be identified: *iterative* modularity and *differentiating* modularity. Iterative modularity refers to using the same kind of module several times, while differentiating modularity refers to the organization of a whole in several differing modules. Self similarity, as seen in fractals, can be seen as a recursive iterative modularity, modules iterating themselves *in themselves*.

Modularity in the brain[1]

The importance of modularity becomes clear when looking at the brain. The brain shows a remarkable modular structure. Its modularity can be described on several levels. The lowest level is of course the subdivision in neurons. But also on larger scales a strong modular structure is present (see also chapter 2). Here it will be argued that *anatomically*, the brain has a highly specific, modular organization, which has strong *functional* implications for cognitive information processing and learning.

Two of the most prominent structural characteristics of the brain will be referred to as *horizontal* structure and *vertical* structure. 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 lines 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. [HUBE62]).

Apart from horizontal structure or a layered structure, there exist at all levels in the primate brain multiple parallel processing pathways that constitute a vertical structuring (e.g. [LIVI88], [ZEKI88]). Vertical structure allows for the separate processing of different kinds of information. A good example can, once more, be found in the visual system, where different aspects of visual stimuli like *form*, *colour*, *motion* and *place*, are processed in parallel by anatomically separate, neural systems, organized in the *magno cellular* and *parvo cellular* pathways (e.g. [LIVI88], [HILZ89]). Convergent structures integrate this separately processed visual information at higher hierarchical levels to produce a unitary percept (e.g. [ZEKI88], [POGG88], [YOE88]). The presence of both horizontal and vertical structure leads to a *modular organization* of the brain [MURR92].

The subdivision of the brain into two hemispheres, as already mentioned in chapter 2, illustrates anatomical modularity at a very large scale. Functionally, this division is paralleled by hemispheric specialization. Whole groups of mental functions are allocated to different halves of the brain. *Split brain* patients in which the connection between the two hemispheres (the corpus callosum) is cut, can live an almost normal life showing that the two hemispheres indeed function to a large extend independently. Within each hemisphere individual functions are again organized anatomically into separate regions. Analysis of behavioral functions indicate that even the most complex functions of the brain can be localized to some extent [KAND85]. For example

[1] Parts of this paragraph are adapted from [HAPP92]. It should be noted that the ideas of this paragraph are not universally accepted by all researchers.

studies of *localized* brain damage reveal that isolated, mental abilities can be lost as a result of local lesions leaving other abilities unimpaired. Warrington described a subject with a severe impairment of arithmetic skills that was not accompanied by a deficit in other cognitive abilities [WARR82]. Also, different types of aphasia (language disorders) indicate that different functions are separately localized in the brain. Patients with *Wernicke's aphasia* are not able to understand written or spoken language while they can speak and write unintelligibly but fluently. *Broca's aphasia* shows the reverse symptoms. The ability of patients to understand language is unimpaired while they are not, or hardly, able to produce language.

An important functional advantage of the anatomical separation of different functions might be the minimization of mutual *interference* between the simultaneous processing and execution of different tasks, needed to perform complex skills like, for example, walking through a forest or driving a car. In psychology, interference studies with multiple task execution indicate that some tasks can easily be performed in parallel, while others strongly interfere. Tasks that are sufficiently dissimilar can be executed simultaneously without much interference. A striking result was reported by Allport, Antonis, and Reynolds who demonstrated that subjects could sight-read music and perform an auditory shadowing task concurrently, without any interference. The simultaneous execution of similar tasks (like presentation of two auditory or two visual messages), causes much more interference. The difference in performance found in these tasks can be explained by assuming a modular organization of the brain [ALLP80]. Some tasks are processed in separate modules and do not interfere. Other tasks require simultaneous processing in single modules and are thus harder to execute in parallel.

In describing the modular organization of cognitive functions in the brain we went from a very large scale grouping of functions into separate hemispheres to a modular organization of individual functions. Evidence indicates that individual functions, again, can be split up into functionally different *subprocesses* or *subtasks*, which, once more, can be localized in anatomically separate regions. Within the field of psychology, for example, *word-matching* experiments show that the human information processing system uses a number of subsequent levels of encoding in performing lexical tasks [POSN86], [MARS74]. In an experiment by Marshall & Newcombe [MARS74], subjects had to decide if two simultaneously presented words did, or did not belong to the same category. Reaction times for visually identical words like TABLE-TABLE were shorter than for visually different words like TABLE-table. Even longer reaction times are found for words that belong to a different category like TABLE-DOG. Posner, Lewis & Conrad [POSN72] explain these data as follows: Visually identical words can be compared on the level of the visual encoding of words. Matching of visually different words can take place when also a phonological code has been formed. Different category words can be compared when an additional semantic code has been formed. These studies indicate that in lexical tasks a separate visual, phonological and semantic encoding analysis of words is involved. PET-scan (positron emission tomography) studies show that these functionally distinguished subprocesses are also separately localized in the brain [POSN88]. Local changes in nerve cell metabolism corresponding to changes in neuronal activity, were registered in subjects during the execution of different lexical tasks. These experiments revealed that visual encoding of words takes place mainly in the occipital lobe in the posterior part of the brain. Phonological codes are formed in the left temporal parietal cortex. Semantic encoding takes place in the lateral left frontal lobe.

An important argument that can be derived from these studies is that the *nature* of information processing in the brain is modular. Individual functions are broken up into subprocesses which can be executed in separate modules (without mutual interference). The modular architecture of the brain might form the necessary neural substrate for the independent processing of different tasks or subtasks. It can be speculated that the subdivision of modules into smaller modules and of functions into sub-functions, might go on into extreme depth. For example, modules containing between 90 and 150 neurons, also known as *minicolumns*, have been proposed as the basic functional and anatomical *modular* units of the cerebral cortex (e.g. [MOUN75], [SZEN77], [ECCL81]). These modules are thought to cooperate in the execution of cortical functions [CREU77].

The question may now be raised at what moment in the development of the brain this modularity arises. An important argument supporting the view that part of the structure is in some way pre-wired, is the anatomical location of different functional regions in the brain. The location of most functions are situated at the same place for almost all individuals, see e.g. [GUYT86]. Another argument to support this view is that most stages in the development of language in children are the same, independent of the language learned. Furthermore all the basic grammar rules are the same all over the world. So clearly, most of the global structure in the brain is already present in the brain at birth, and, as a consequence, has to be genetically coded in some way. This has to be a modular coding, because the genes do not have enough capacity to store all specific connections.

Modularity in genetics

The genetic coding of all life forms on earth is also (very) modular. All genetic information is stored in genes, which are contained in long double-stranded helical strings, the DNA, of which each cell in an organism has a copy[1]. 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* (see figure 1, with * as 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,

	U	C	A	G	
U	phe	ser	tyr	cys	U
	phe	ser	tyr	cys	C
	leu	ser	*	*	A
	leu	ser	*	trp	G
C	leu	pro	his	arg	U
	leu	pro	his	arg	C
	leu	pro	gln	arg	A
	leu	pro	gln	arg	G
A	ile	thr	asn	ser	U
	ile	thr	asn	ser	C
	ile	thr	lys	arg	A
	met	thr	lys	arg	G
G	val	ala	asp	gly	U
	val	ala	asp	gly	C
	val	ala	glu	gly	A
	val	ala	glu	gly	G

Figure 1. *The Genetic Code of RNA (from [HOFS79]).*

[1] *Although there are life forms containing only RNA.*

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 each cell will operate. It is supposed that during the growth of an embryo a process of *cell differentiation* takes place that is caused by the forming of certain proteins that generate a positive feedback on the genes in the DNA that produced those proteins. These proteins will also repress another group of genes and once the positive feedback has started, it will never stop, so the repressed group of genes will never be active again. Embryological experiments show also that certain cells in an embryo control the differentiation of adjacent cells [GUYT86], hereby implementing the idea of *context*. Most mature cells in humans only produce about 8000 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 detail.

As argued in the previous paragraph, the brain is supposed to have an initial structure at birth. 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).

How modularity is coded

In addition to the already mentioned minicolumns, the cortex apparently contains, at a higher level, yet another form of modular organization. So called *macro modules* consist of an aggregation of a few hundred minicolumns, forming a larger processing module. According to Szentagothai [SZEN77]: '*the cerebral cortex has to be envisaged as a mosaic of columnar units of remarkable similar internal structure and surprisingly little variation of diameter*'.

This kind of iterative modularity can be seen almost everywhere in nature:

- leaves of a tree,
- alveoli in the lungs,
- scales of a fish,
- rods and cones in the eye,
- minicolumns in the brain,
- hairs on the skin,
- ants in a colony,

and so on. It suggests that iterative modularity is a very common principle in nature. In the process of evolution it has time and again been profitable for all kinds of species to *duplicate* that which already has been 'invented' once before. An example of this is found in [DAWK86]: The middle part of the body of a snake is composed of a number of *segments*. Each segment consists of a vertebra, a set of nerves, a set of blood vessels, a set of muscles etcetera. Although each individual segment has a very complex structure, it is exactly similar to other segments. Therefore, in order to add new segments all that has to be done is a relatively simple process of duplication. The genetic code for building one segment, which is of great complexity, is the result of a long and gradual evolutionary search. However, since the genetic code for building one segment is already present, new identical segments may easily be added by a single mutational step. '*A gene for inserting extra segments may read, simply, **more of the same here**.*' Snakes have many more vertebrae than their fossil ancestors as well as their living relatives. '*We can be sure that during the evolution of snakes, numbers of vertebrae changed in whole numbers. We can not imagine a snake with 26.3 vertebrae, it either has 26 or 27.*' This example shows that complex, partial architectures (read: modules) once encoded during evolution, can be used repeatedly to produce the whole system. So apparently genetics is not only on a small scale based on modules, but it has the effect of generating modular structures at larger scales as well.

The conclusion that may be drawn from this iterative nature of the translation from genes to organisms (or: from *genotype* to *phenotype*) is an important one: the genetic code does not describe exactly what the final form of an organism will be, but describes a number of rules that, when followed, will *result* in the final form. In other words: instead of a *blueprint* the genes contain a kind of *recipe* [DAWK86]. This means that there is no one-to-one correspondence between a part of the DNA and a part of the actual organism. It allows the genetic search to utilize already discovered principles repeatedly. Not the resulting organisms, but the recipes that built them are combined in evolution. Only the rules (proteins) that are able to work together in the forming of a fit organism will survive.

This is probably what has happened in the case of the human brain. The human brain is the result of an evolutionary process, during which the brain (in our predecessors) became larger and larger. If the human brain is compared with the brain of, for example, a cat or a macaque monkey, the only difference is the size, which is in the case of the human significantly larger, in particular the cerebrum (the largest part of the brain consisting of the left and right hemisphere) which is responsible for all higher cognitive functions. This ‘size’ has of course to be corrected for the size of the animal. The brain of an elephant is much larger, for example. (See [JERI85] for a measure of brain sizes: the *encephalization quotient*.) The cortex contains, as mentioned, a high amount of repeating modules. The only thing that was necessary, as was the case with the segments of the snake, were genes saying ‘*more of this, please*’.

Exactly how the coding of structure in organisms is actually done in nature is mostly unknown, and is still an active area of research.

Imitating the evolution of the brain

Modularity is an important principle in nature and, as argued, is to a large extent present in the brain. The question is whether this modularity, when used to construct *artificial* neural networks, will result in better performances. Preliminary results show that this may be the case (e.g. [RUEC89],[SOLL89],[MURR92],[HAPP92]), but so far no results are available concerning large artificial neural networks with a modular structure. The problem is that the operation of neural networks is very complex: even with a well known algorithm like backpropagation it is very difficult, if not impossible, to understand exactly what happens inside, particularly when more hidden layers are used. It becomes even more difficult when *recurrent* networks are used, these are networks without the feedforward restriction of the backpropagation algorithm (see chapter 2). The behaviour of these networks can only be described by a large number of coupled differential equations, of which no solution exists in practice. This has large consequences for the *design* of neural networks. It is not possible to calculate in advance what the optimal topology of a network should be, given a specific task. The only way of determining the quality of a given network for a specific task is *testing* it. The only description of the behaviour of a network is its simulation.

But there *is* a large scale neural network that functions rather well and is based on modularity: the brain. And as in chapter 2, where was explained how reverse engineering of the brain led to the ‘invention’ of artificial neural networks, reverse engineering can also be used to find a method that is able to find good modular structures for artificial neural networks. The process that ‘invented’ the brain is, of course, evolution. In millions of years the process of evolution gradually resulted in the increased complexity of an aggregation of information processing cells, the brain. Reverse engineering of this process logically leads to the use of a genetic algorithm, that, as described in chapter 3, is a simulation of evolution.

The use of genetic algorithms relieves us of a problem mentioned above: the impossibility of describing why one network performs better than another. Genetic algorithms do not care why one solution is better than the other, they are not even *able* to use such information. The only drive behind its reproductive functioning is the fitness of the members in its population. This fitness can easily be calculated. Just generate the neural network represented by one of the members of the population, and *test* it.

There are several ways in which a genetic algorithm can be used to find neural network solutions. The main two possibilities are:

- Use the genetic algorithm to find the weights of a given network structure that result in the smallest error. With this method, the genetic algorithm is used instead of a learning algorithm like backpropagation. The genes of the algorithm have a one-to-one correspondence to the weights of the network. A slight variation of this method is to use the genetic algorithm to find a set of reasonably good weights, leaving the *fine tuning* to a learning algorithm (see for example [WHIT89B] and [GARI90]).

- Use the genetic algorithm to find the structure of a network. With this method the genetic algorithm tries to find the optimal structure of a network, instead of the weights of a given structure. The genes of the genetic algorithm now contain a coding for the topology of the network, specifying which connections are present. The weights of the network have to be trained as usual (see for example [DODD90]).

Of course also combinations of the two methods are possible: coding the presence of connections as well as their weights in the genes. But shown by the XOR example in chapter 2, with the input and output layer directly connected, given a good topology the network should always converge, so the training of the weights can better be done using a learning algorithm instead of a genetic algorithm.

Most of the work done on finding good network topologies with genetic algorithms used a kind of *blueprint* method of coding the topology in the genes of the members of the population (e.g. [HARP89],[MARI90]). But because nature does *not* code for blueprints, a research project trying to construct a method that enables the coding of *recipes* in genes was started [HAPP92].

Using graph grammars as recipes

A neural network, when seen as a collection of nodes and edges, is a *graph*. So what was needed to code neural network structures was a method of *graph generation*. There are several formal languages describing graphs, some rewrite nodes, others rewrite edges, others perform operations on adjacency matrices, but a more logical choice for this research was again found in biology. To describe the form and the growth of plants, the biologist Aristid Lindenmayer made a mathematical construct called L-systems (see chapter 4), to describe the development of multicellular organisms. The method takes genetic, cytological and physiological observations into account as well as purely morphological ones [LIND68]. Since the biological metaphor is one of the mainstays of this research, taking L-systems for our recipes was an obvious choice (the original idea coming from [HAPP92]). Besides, L-systems offer the possibility of describing highly modular structures. They are very good at describing (giving a recipe of) both iterative and differentiating modularity, and are often used to describe fractals. Other graph grammars were shortly investigated, but they seemed to lack the flexibility offered with L-systems. One of the major advantages of L-systems over usual graph grammars is the ease of including *context*, enabling the analogy of cell differentiation in the 'growth' of the neural network.

L-systems are however not specifically designed to describe graphs. It is just a string rewriting method. The actual meaning of the resulting strings depends on the interpretation of the symbols used. In chapter 6 will be explained which symbol interpretation was used, and how the *production rules* were coded genetically.

Combination of metaphors

In this research we tried to combine three methods with their origin in biology:

- Genetic Algorithms
- L-systems
- Neural Networks

Our goal was to design a method that searches automatically for optimal modular neural network architectures. In this chapter it was argued that the concept of modularity is frequently used in nature, and results in good solutions. Our working hypothesis throughout this research has been that by making use of modular techniques based on biological metaphors, both in search method and in network structure, the eventual performance of our method for finding optimal neural network structures would be better than other methods.

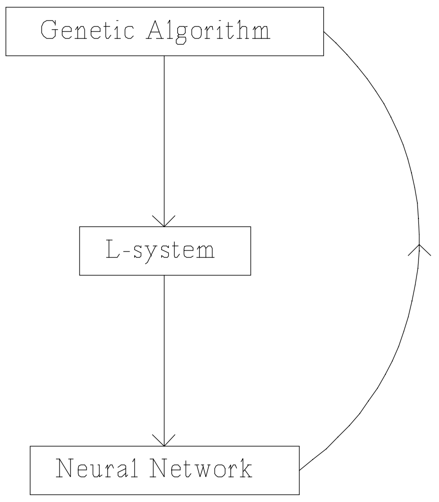


Figure 2. *Global structure.*

Our method can be summarized as follows:

1. A genetic algorithm generates a bit string, this is the chromosome of a member of its population. The search of the genetic algorithm is directed towards members 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 (in this research backpropagation) trains the resulting network structure for the specified problem. The resulting error is then transformed into a measure of fitness: a low error results in a high fitness. This fitness is returned to the genetic algorithm.

This method combines the theory explained in the last three chapters. The next chapter describes in detail the transformation from genotype to phenotype.

‘*Cheer up, the worst is yet to come.*’
— Philander Johnson

6 The grammar and its coding

This chapter describes a number of L-system grammars that can be used to define network structures. Besides the L-system actually used, some other possible codings we looked at are described too. In the last paragraph the coding of the rules into the chromosomes is described.

From network to string

As described in chapter 5, our GA has to manipulate a population of sets of production rules. Each member of the population is a binary string consisting of one or more production rules for an L-system. To determine the fitness of a string, the production rules are extracted from the string and an L-system rewrites an axiom using those rules. The resulting string is interpreted as a network to be trained by (in our case) backpropagation. Using backpropagation restricts the possible topologies to *feedforward* networks: all the nodes can be indexed in such a way that there are only connections from n_i to n_j with $i < j$.

One of the first ideas was to represent each node in the network with any letter from the alphabet and use extra symbols (`[,]` and digits) to implement modularity and connections.
A variation of this coding used *implicit* connections between neighbouring letters and the extra symbols for special connections. The comma was introduced to note that two neighbouring letters were not connected (as in `A,B`). Letters could be grouped using the `[]` symbols. Each pair of brackets was associated with a number (or level) and connections to these groups could be made by using that number. These strings, which will be called *absolute pointer strings*, are described later.

Another idea restricted the alphabet to `1`s and `0`s representing nodes in a network at various levels. Using a connection table, forward connections were made between specific combinations of `1`s and `0`s. Modules are introduced by placing nodes at different levels in a string. The nodes within a module were connected to nodes at the same level according to the connection table. The resulting strings, called *binary table strings* are described in detail later.

The last strings that will be described are the ones actually used in our research (called *relative skip strings*). They don't have the numbered brackets (although they do have the brackets themselves) and use digits to indicate a relative jump within the string in order to make a specific connection. Two variations are possible: neighbouring characters are connected by default and a comma is used to denote the opposite, or neighbouring characters are not connected by default and a minus symbol is used to indicate a connection between neighbouring characters.

Example networks

This paragraph presents a number of connection structures that we used to evaluate the string notations. For each of these structures (and many others) we tried to find a string representation, in order to test the representational potential of the used grammar.

Basic networks

Figure 1 shows four basic elements that were used to evaluate the strings. 1a and 1b are the simple elemental structures whereas 1c and 1d combine these elements introducing modularity which should be exploited by the grammar.

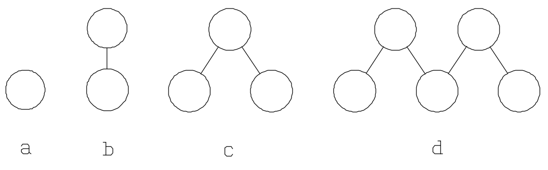


Figure 1. *Four basic elements.*

XOR

Figure 2 shows two networks that are able to solve the XOR problem (exclusive OR - see chapter 2).
The extra connection in 2b results in better performance, although often more complicated strings were needed to code it.

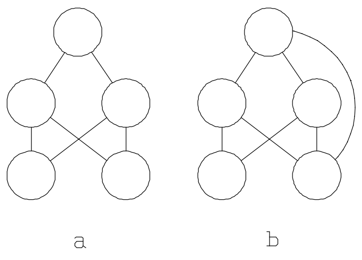


Figure 2. *Two possible networks for solving the XOR problem.*

Absolute pointer strings

With absolute pointer strings, nodes of the network are represented by characters from the alphabet. The same character can be used for all nodes, but if more characters are used, specific contexts can be used. If characters are placed within [ ]'s the nodes they represent are said to form a *module* and they can be referred to as one group of nodes in other parts of the string. Each pair of [ ]'s has an index associated with it (denoted as $[_i\ ]_i$). By appending a digit pointer to a node (or module), a connection is coded from that node (or module) to all modules of which the index corresponds to that pointer. Connections between neighbouring nodes (or modules) are made automatically unless they are separated by a comma. Only feedforward connections are made by connecting nodes and modules to nodes and modules to their right in the string.

Basic networks

The basic elements shown in figure 1 are easy to represent using the grammar described. Figure 1a can be represented by a single character (A for example) and figure 1b by two characters (for example AA or AB). A1,A$[_1$A$]_1$ is a possible string encoding the network of figure 1c, where the first A is connected to the third A, but not to the second. There is a much better, smaller solution: [A,A]A. The fourth example from figure 1 is more difficult: [A,A1]A,A$[_1$A$]_1$ is one of the simplest representations.

XOR networks

Since the XOR network of figure 2b is a simple extension of the network shown in figure 2a, it will be convenient to be able to use the string representing figure 2a for creating a string representing figure 2b. Because two modules are always fully connected, 2a can easily be written as [A,A][A,A]A. To make the extra connection in figure 2b, just one pair of brackets and a digit have to be added: [A,A1][A,A]$[_1$A$]_1$.

These strings are able to model all feedforward networks because every single node can be put between numbered brackets so digits can be used for all connections. Below however, are some considerations that made us decide not to use the grammar in the described form because we were looking for simple strings and production rules.

- The L-system should be able to generate the numbers of the brackets automatically.
- Production rules should be able to rewrite a specific module (referred to by pointer) as well as all modules with a specific content and thus the pointers should be matched against some sort of *wild card*, which requires changes to the L-system.
- Removing a pair of brackets with a production rule also requires the corresponding pointers to be removed. This can not be done with a production rule and therefore requires changes to the L-system.

At the same time when we were experimenting with this grammar, we looked at the binary table strings, which use a binary alphabet.

Binary table strings

In binary table strings, nodes are represented by 1s and 0s. When a bit is rewritten brackets are automatically placed and are as such not present in the production rules. Nodes between brackets are said to be at a different level. The brackets are indexed with 0 or 1, depending on the character they originated from.

Connections are made depending upon a *connection table*. Using the values of the characters as indexes in the connection table, a character is connected with all characters to its right (ensuring a feedforward connection) for which the Connect? column contains a 1. In the string 0110, using the connection table from figure 3, the first 0 is connected to both 1s, but not to the last 0. The first 1 is connected to the last 0, as is the second 1. For the rest of this paragraph, the connection table shown in figure 3 is used.

From	To	Connect?
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3. *Sample connection table*

Context plays an important role, only single bits are used as predecessors and depending on their context they are rewritten by a successor string which is surrounded by brackets indexed with the original character. Using the production rules `0 > 0` $\rightarrow$ `10` and `1` $\rightarrow$ `11`, the string `001` will then be rewritten as $[_{0}10]_{0}0[_{1}11]_{1}$ after one step. This string can be visualized by the network in figure 4, where the modules are highlighted in rectangles. The bottom rectangle represents the `10` sub-string and the top rectangle the `11` sub-string. The fifth node represents the single `0`.

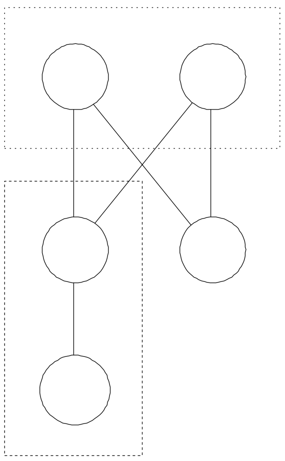


Figure 4. *Visualization of binary string.*

Figure 4 also shows how connections are made when modules are present. First is decided which nodes and modules *may* be connected by looking at the `1`s and `0`s at the outermost level (the nodes that are not within brackets and the numbers below the outermost brackets). In our example, the `10` module has number `0` and may be connected to the `11` module (which has number `1`), but not to the `0`. The single `0` may also be connected to the `11` module.

To determine which nodes are actually connected with nodes in the other module, we look at the next level. In our example, the `1` and `0` of the `10` module are compared against the `1`s in the `11` module. Because `1-1` has a `0` in the third column of figure 3, only the second node (the `0`) is actually connected to the nodes in the `11` module. The `0` that is not within brackets is connected to both `1`s from the `11` module because we are at the lowest level for the `0` (there are no more brackets). This applies to all nodes/modules: if there are no more levels to compare, the remaining nodes/modules are fully connected.

This process of comparing numbers using the connection table is repeated for all remaining levels and modules.

Basic networks

Figure 1a can be represented by either a `0` or a `1` and figure 1b by either `01` or `10`. The third element from figure 1 can for example be written as `001` or `110`. The fourth example is more difficult: we need multiple levels to code the network. One of the simplest possibilities is $[_{1}0]_{1}[_{0}01]_{0}[_{0}01]_{0}$. In order to reconstruct the network from this string, we first note that the first module may be connected to the second and third module (using the numbers below the brackets), but the second not to the third. To determine which of the nodes actually are connected to the second and third module, each of them is compared against the nodes from those other modules (again using the connection table). Finally, the nodes within the second and third module are connected.

XOR networks

The first XOR network from figure 2 was easy (00110), but we were not able to find a string representing figure 2b using the connection table from figure 3. However, when we used the connection table shown in figure 5, the solution was not too difficult: $[_1 01]_1 01[_0 1]_0$. We also tried figure 2a again using this table. The found solution is not as simple as the first one: $[_1[_0 01]_0]_1 01[_0 1]_0$.

From	To	Connect?
0	0	1
0	1	0
1	0	1
1	1	1

Figure 5. *Connection table used to represent for figure 2b.*

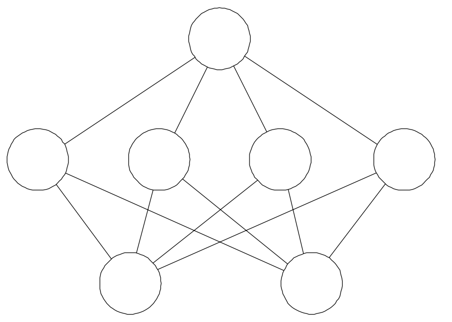


Figure 6. *Network with 2 input nodes, 1 output node and a hidden layer of 4 nodes.*

Another network tried is shown in figure 6. Coding it was easy using the table from figure 3 (0011110), but using the table from figure 5 resulted in

$$[_1[_1[_1[_0 01]_0]_1]_1]_1\underline{0[_1 0[_1 01]_1]_1}[_0[_0[_0 1]_0]_0]_0$$

The hidden layer is underlined.

When we had to decide which type of grammar to use in our research we still had some doubts about the possibility to code *all* possible feedforward networks with the binary table strings.

Also, because the production rules only contain 1s and 0s (the brackets are inserted automatically) complete modules (including the brackets) can not be rewritten into smaller modules. At the time of writing we had discovered a method to code all feedforward networks using the table shown in figure 7. The problem with rewriting modules still remains.

From	To	Connect?
0	0	1
0	1	1
1	0	1
1	1	0

Figure 7. *A NAND style connection table.*

For the last type of strings we tried, we returned to a normal alphabet (A-Z) and some of the ideas used with the absolute pointer strings.

Relative skip strings

The strings actually used in our research are made of characters from the alphabet {A-Z,1-9,[,]} ∪ {,}. A node from the network is represented by a letter from our alphabet (A-Z). Two adjoining letters are automatically connected feedforward. If two letters are separated by a comma, no connection is made. Modules can be created by grouping nodes (or other modules) between square brackets. In contrast to the multilevel strings, brackets don't have indexes attached. Two adjoining modules are *not* fully connected. Instead, all *output nodes* from the first module are connected to all *input nodes* from the second module. An input node is a node that receives no input from within the module. An output node has no output to other nodes within the module. This specific connecting of modules facilitates the combining of independent networks into one larger network. As with single characters, a comma is used to prevent the connection of two adjoining modules.

In order to make specific connections between nodes or modules that are not side by side in the string, single digits are used to denote a *skip* within the string. When a digit *x* is encountered, it should be interpreted as "skip *x* nodes and/or modules to your right, and make a connection to the next node or module". So within the string `A1BC` the `A` is connected to `C` (the 1 skips the `B`) but also to `B` (because there is no comma between the `A` and `B`). In `A1,BC` or `A,1BC` the `A` is only connected to the `C`.
Modules count as one while skipping, so in `A1,[BC]D` the `A` is connected to `D`, *not* `C`.

If a skip is contained in a module and goes beyond the closing bracket, the skip is continued after the `]`.
In the string `[A2[B,C]D]E`, `A` is connected to `B` and `C` because both are input nodes in the `[B,C]` module (if there is no comma between `B` and `C`, `A` would not be connected to `C`). `A` is also connected to `E` because the 2 skips the `[B,C]` and the `D` and connects to `E` (even though `E` is outside the module containing `A`). The network coded by this string is shown in figure 8.

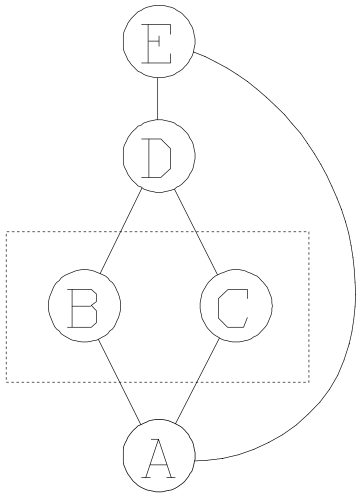


Figure 8. *The string [A2[B,C]D]E visualized.*

Basic networks

Using the connection method described above, the coding of the networks from figure 1 is easy:

Figure 1a:	`A`	(or any other letter from `A-Z`, only `A` is used in the examples)
Figure 1b:	`AA`	
Figure 1c:	`[A,A]A`	(both `A`'s in `[A,A]` are output nodes)
Figure 1d:	`[A,A2]A,AA`	

XOR networks

The XOR networks too, are not very difficult:

Figure 2a:	`[A,A][A,A]A`	
Figure 2b:	`[A,A1][A,A]A`	(note the similarity between the two strings)

The network from figure 6 with 4 hidden nodes can be written as:

`[A,A][A,A,A,A]A`

As can be seen in this string, in order to generate layers of internally unconnected nodes (modules), production rules should be generated that insert commas into the string.
Because we wanted modular networks with unconnected blocks of nodes to develop as easily as possible, we also tried a variation that does not connect neighbouring nodes (a minus symbol or zero skip needs to be explicitly specified to do so).

The strings encoding figure 1c and 1d then become `[AA]-A` and `[AA2]-AA-A` respectively. The XOR networks can be written as `[AA]-[AA]-A` and `[AA1]-[AA]-A`. Finally, the network from figure 6 can be written as `[AA]-[AAAA]-A`.

Production rules

The L-system used for generating the strings described in the previous paragraph is a 2L-system: every production rule can have both left and right context. The production rules have the same format as described in chapter 4:

```
L < P > R → S
```

The four parts of the production rule can not be arbitrary strings over the alphabet {A-Z,1-9,[,]} ∪ {,}. The following restrictions apply to the different parts:

Predecessor

The predecessor (P) may only contain complete modules and nodes. Therefore the number of left and right brackets must be equal and in a correct order. Each module must be complete and thus the following string (for example) will be discarded: A]BC[D. Also, the predecessor may not contain empty modules ([]). Finally, the predecessor should at least contain one letter (node).

Successor

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

Contexts

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 4 is the handling of the context. Whereas in the L-systems from chapter 4 the context is matched against the characters to the left and/or right of the predecessor, in the L-system used, the context is compared with the coding characters of the nodes from which one or more nodes in the predecessor receive their input (left context), or the nodes connected with the output from one or more nodes in the predecessor (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 with 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.

For an example let us look at the production rules shown in figure 9. If we take A as axiom, the rewriting process can be described as follows:

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

Figure 9. *Sample production rules.*

First we have the axiom A (figure 10a).
After one rewriting step we have the string BBB (figure 10b).

During the second rewriting step, the first B is rewritten using rule 2 because the first B is connected to the second (figure 10c). The second B is also rewritten using rule 2 (figure 10d). The third B is rewritten using rule 3 because the third B is not connected *to* another B (there is only a connection *from* another B). This finally results in the string `[C,D][C,D]C`, shown in figure 10e. Because all rewriting goes on in parallel, the strings for figures 10c and 10d are not created separately, but figure 10b is directly rewritten into 10e. The other figures are there for clarity.

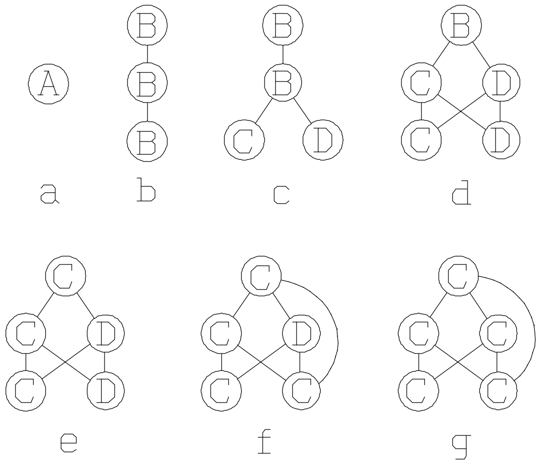


Figure 10. *The rewriting of A with rules from figure 9 visualized.*

During the third rewriting step the first D is rewritten using rule 5 because the first D is connected to the second D (figure 10f) and the second D is rewritten using rule 4 because the first C is connected to the second D.
No more rules apply, so the final string is `[C,C1][C,C]C`, shown in figure 10g. Again, figure 10f is just for clarity, in reality figure 10e is rewritten in parallel into figure 10g.

Coding of production rules

As described in chapter 3, genetic algorithms work with coded parameters of a problem. In our research, the parameters represent a set of production rules. This paragraph describes the coding from production rules to binary strings that can be used by a GA.

A production rule consists of four (possibly empty) parts. Each of these parts is a string over the alphabet {A-Z,1-9,[,]} ∪ {,}. Although this particular alphabet implies using 26 different letters, any number is possible (by using other symbols for the nodes too). The same goes for the number of skip symbols. We choose (rather arbitrary) to use 8 different letters (A-H) and 5 different skips (1-5). This brings the total numbers of characters in our alphabet to 16 (8+5+2+1).

There are several ways to code a rule, but we decided to represent each character from the rule by a fixed-length binary string. A binary string of length l can be used to code 2^l different characters. All 2^l binary codes must be assigned to one of the characters of our alphabet. Both the number of codes (determined by l) and a distribution had to be chosen.

The length and distribution we choose are loosely based upon the biological *genetic code* which uses a four character code (see inset). Each of the four different bases used in the genetic code is coded binary (using two digits) and therefore, each triplet of bases is in our translation replaced by a binary string of length 6.

The 64 possible strings of length 6 were then distributed amongst 17 symbols. The 17 symbols include the 16 characters from our alphabet plus a special symbol (an asterisk), used to separate constituent parts of production rules (context, predecessor, successor) from each other within a chromosome. The asterisks can be compared to the start and stop markers used for the transformation from RNA to protein.

The resulting table is shown in figure 11 (the original genetic code table can be found in chapter 5).

The table is read as follows: to determine the character corresponding to a bitstring, determine which of the four rows on the left corresponds to the first two bits of the string. Then, choose a column according to the middle two bits, and finally choose a row on the right using the last two bits. For example, the character corresponding to `100100` is the first `A` in the table.

The other way around, from character to bitstring is also easy: the bitstring is the concatenation of the two bits to the *left*, the two bits *above* and the bits to the *right* of the character. So the bit string for character `5` is `001111`.

As mentioned above, an extra character (the asterisk) is used to separate the four parts of a single production. The following rules are followed in order to extract a production rule from the chromosome:

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

Figure 11. *The conversion table used.*

1. Pick a starting point anywhere in the bitstring (chromosome). Start reading the L part of the production rule.

2. Do this by reading the bits, six at a time. Look up the character corresponding to the six long bitstring using the table from figure 11.

Besides controlling the development and final form of a biological organism, genes also control the reproduction and day-to-day function of all cells. Each gene, a nucleic acid called *deoxyribonucleic acid (DNA)*, automatically controls the formation of another nucleic acid, *ribonucleic acid (RNA)*, which spreads throughout the cell and controls the formation of specific proteins. These proteins control the function of the cells.

Genes are contained, large numbers of them attached end on end, in long, double-stranded, helical molecules of DNA. The two strands of the helix are held together by four kinds of *bases* (which names can be represented by the four letter alphabet *A, T, G* and *C*) and a *hydrogen* bonding. Because the hydrogen bonding is very loose, the two strands can easily split up. If it does so, the bases of each strand are exposed. These exposed bases form the key to the so called *genetic code*. Research studies in the past few years have demonstrated that this genetic code consists of successive ’triplets’ of bases - that is, each three successive bases are a code word. These words control the sequence of amino acids in a protein molecule during its synthesis in the cell. The bases can be ‘read’ in both directions, and reading can start at any position, not only from positions 1, 3, 6 etc.

Because the DNA is located at the nucleus of the cell and most of the functions of the cell are carried out in the *cytoplasm* (which surrounds the nucleus), the information from the DNA is *transcribed* onto RNA, which it then transports to the cytoplasm. The base triplets from the DNA are transcribed into base triplets on the RNA. The RNA also contains four bases (abbreviated as *U, A, C* and *G*). Each of the (4x4x4 = 64) base triples of the RNA is then translated into one of twenty amino acids or a ‘start protein building’ or a ‘stop protein building’ signal. The conversion table used for this is also called the *genetic code*.

A short introduction to the genetic code (adapted from A.C. Guyton’s Textbook of Medical Physiology).

3. If the character is not an asterisk, add the character to the string read so far.

4. Otherwise, assign the string read so far (which can be empty) to the current part of the production rule (L,P,R or S). Advance to the next part of the production rule. If the last part read is S, start reading the next production rule.

5. Repeat steps 2-4 until all bits from the chromosome are read. If no more bits are present to complete the current production rule, discard this production rule.

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 to the asterisk in our chromosomes), 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 with the *same* genetic information. The different proteins are build by adjusting the starting point for reading by just one base compared to the starting point of the other proteins, instead of three bases needed for one code word. The genetic information can be seen as a number of *overlapping* recipes for different proteins. In order for our chromosomes to contain this kind of overlapping information, our algorithm can start at *any* bit position. As a result of this the chromosome can be read *twelve times* by following the above rules first with the starting point at bits 0 to 5 respectively, reading the string *forwards*, and then by following the rules with the starting point the last six bits respectively, reading the string *backwards*. Because of the high level of information contained in our chromosomes, the level of implicit parallelism may eventually be significantly higher than with a genetic algorithm using strings which are read only once.

Example

A (small) example is shown in figure 12, although in our research we often used chromosomes of length 512 or longer.

In the example, a bitstring is shown together with four translations. The same string also contains six other translations which are not shown for reasons of clarity.

The four translations shown are: (taking into account the direction of reading)

```
*2][[]A*
,H[2[D,
*A*BBB*
,*2]]C1,
```

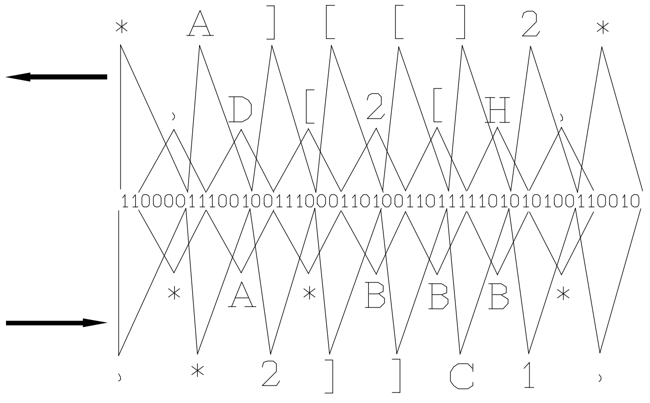


Figure 12. *Extract from a chromosome together with 4 translations.*

None of these strings forms a complete production rule on its own, but if we imagine the third string being part of a longer string, for example:

```
F1,**A*BBB*C*A]],
```

(where the **bold** part is the third string), the underlined part can be seen as a complete production rule (it contains five asterisks). If we write the production in our usual notation, we get:

```
A > BBB → C
```

Repair mechanism

Using chromosomes of length *l* results in $2 \cdot 6 \cdot \left\lfloor \frac{l}{6} \right\rfloor$ characters (which means more characters than the total number of bits). 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-character 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 as a starting point for a production rule (the last four markers placed at the end of all ten strings, can not be used). If we discard all production rules that contain invalid character combinations (those described earlier), we are left with much less production rules.

With real DNA, in the few hours between DNA replication and mitosis (the actual process of a cell splitting into two new cells), a period of very active repair and ‘proofreading’ of the strands starts. Special enzymes cut out defective areas and replace them with appropriate nucleotides. Because of this, almost never mistakes are made in the process of cell replication (if it does happen, it is called *mutation*).

The developed software also contains functions to repair faulty strings. The functions remove spare brackets, commas, digits, etc. in order for the string to meet the restrictions from the previous paragraph. The functions are described in chapter 7. The number of usable production rules after this ‘repair’ process is usually about 50.

Hofstadter's Law: '*It always takes longer than you expect, even when you take into account Hofstadter's Law.*'
— Douglas R. Hofstadter

7 Implementation

The software written and used during our research consists of a number of separate programs: a main module (`GENALG`) which controls the population of production rules, and three sub programs. The first of these (`CHR2GRAM`) translates a member from the population into a set of production rules. The production rules are then rewritten using the second program (`LSYSTEM`), which has an adjacency matrix of the network as its output. Finally, the network represented by this matrix is trained using backpropagation (`BACKPROP`).

This chapter describes the software. All programs were written in C, except for the back-propagation module, which was written in C++. Some elements are described in detail, while other parts are just briefly mentioned.

Environment

All the software has been written to run on both MS-DOS computers as well as Unix based machines. During our research we had two PC's (an i386 with 387 and an i486) at our disposal. Once the software was more or less finished we started porting it to the Sun network at the department of Computer Science at Leiden University. That network, consisting of several Sparc4 based Sun's (ELC, LPC and IPX models) provided us with the parallel computing power needed for the larger simulations. However, Murphy's Law also liked that network, and during our testing and simulations, the network was down regularly. This was due to internal reorganizations, bad cabling, full hard disks and no system operator for the network (of which three items have been resolved at the time of writing). The final version of the software can easily be ported to other computers (for example a network of transputers) wich have both C and C++ languages available.

Extended GenLIB

The GenLIB library mentioned in chapter 3 was rewritten so chromosomes with length longer than 16 bits could be used.

The new version, called *Extended GenLIB*, also includes functions to manipulate the population on disk, instead of in memory, because with long chromosomes the total population size (in bytes) can become very large. Next, the main datastructures and all the functions from the library are described shortly. Three functions specific to our research are described also in detail.

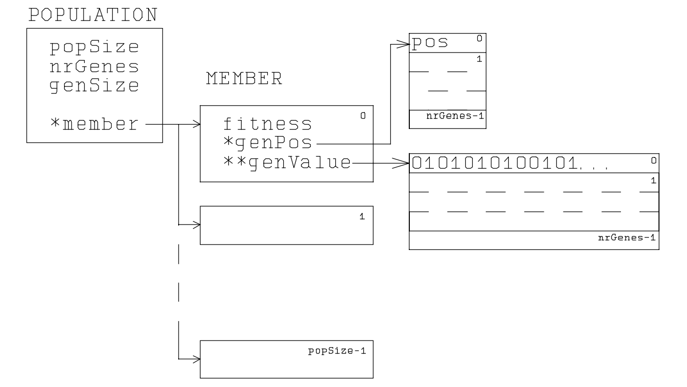


Figure 1. *Overview of the main datastructures.*

Datastructures

Each population is described by a `POPULATION` structure:

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

   MEMBER   *member;
} POPULATION;
```

The pointer to `MEMBER` is an array of `MEMBER` structures:

```
typedef struct     /* contains information for each member */
{
   float fitness;    /* fitness of the member              */
   unsigned *genPos; /* pointer to array of genpositions   */
   BYTE **genValue;  /* pointer to array of chromosomes    */
} MEMBER;
```

Each `MEMBER` structure contains pointers to an array of genpositions (used by inversion) and an array of bytes containing the actual chromosome. In order to keep the source as simple as possible, only chromosomes with a length which is a multiple of 8 (the number of bits in a byte) are allowed.

When the population is saved to disk, the file starts with a header:

```
typedef struct        /* this structure is the header of a
                         populationfile on disk           */
{
   unsigned long generation;   /* generation number
                                  (parameter of Save...()) */
   unsigned popSize,
            nrGenes,
            genSize;
} POPFILEHEADER;
```

Then, for each member, the fitness, position array and bitstring are saved.

Functions

The library is divided in several modules, where each module contains functions for a specific task. Next each of the modules will described together with its functions.

The initialization module

This module contains functions for allocating memory for either a whole population or just one member (useful when memory is short).

```
POPULATION *DefinePopulation(unsigned popSize,
                             unsigned nrGenes,
                             unsigned genSize,
                             BOOLEAN  initialize);

MEMBER *DefineMember(unsigned nrGenes, unsigned genSize);
```

The `initialize` parameter is used to indicate whether the bit strings should be randomly initialized or cleared to zero.
Depending on which of the functions is used to allocate the datastructures, one of the following functions is used to free the allocated memory:

```
void FreePopulation(POPULATION *p);
void FreeMember(MEMBER *m, unsigned nrGenes);
```

Storing and retrieving the members

Functions are available to save and load whole populations from disk, either by name or by reference to a file pointer.

```
int SavePopulation(POPULATION *p, FILE *fp,
                   unsigned long generation);
int SavePopName(POPULATION *p, char *fileName,
                unsigned long generation);
int LoadPopulation(POPULATION *p, FILE *fp,
                   unsigned long *generation);
int LoadPopName(POPULATION *p, char *fileName,
                unsigned long *generation);
```

The `generation` parameter is stored (or retrieved) in the file, and can be used as a kind of progress indicator.

If memory is short, the members can manipulated directly on disk with the following functions:

```
int SaveMember(MEMBER *m, unsigned nrMember, FILE *fp,
               unsigned nrGenes, unsigned genSize);
int LoadMember(MEMBER *m, unsigned nrMember, FILE *fp,
               unsigned nrGenes, unsigned genSize);
```

The `nrMember` parameter specifies which member to save or load.

Selection and replacement

Both selection methods described in chapter 3 (roulette wheel and rank based) are available using the functions

```
     unsigned Select(POPULATION *p);
     unsigned RankSelect(POPULATION *p, double pressure);
```

The `pressure` parameter indicates how much larger the probability is of the highest ranked member being selected than the member in the middle of the population. E.g. with a pressure of 2.0, the top ranking member in a population of 100 members has a twice as large probability of being selected than number 50. This function is the same as proposed by Whitley [WHIT89A] and accepts pressure values between 1.0 and 2.0.

For rank based selection to work, the population must be sorted. This can be done with the quicksort algorithm using:

```
     void SortPopulation(POPULATION *p);
```

The one-at-a-time replacement described by Whitley can be done using

```
     unsigned RankReplace(POPULATION *p, POPULATION *np,
                          unsigned newmember);
```

The genetic operators

The genetic operators described in chapter 3 can be applied with the following functions:

```
     void Mutate(POPULATION *pop, unsigned member,
                 int variance, double pMut);

     int Crossover(POPULATION *oldpop,
                   unsigned parent1, unsigned parent2,
                   POPULATION *newpop, unsigned child,
                   double pCross, unsigned points);

     int Invert(POPULATION *pop, unsigned member, double pInv);
```

The `pMut`, `pCross` and `pInv` parameters set the probability of the operator being applied. For `Crossover()` the member to receive the crossover result can be specified, the other functions operate on, and change the parent bitstring.

Finally, a function is available that generates a complete new population using roulette wheel selection, crossover, inversion and mutation:

```
     int Propagate(POPULATION *p, double pInv, double pMut,
                   double pCross, unsigned int points);
```

A sample program using one-at-a-time selection and replacement is given below. The program calls a function `Fitness(POPULATION *p, unsigned member)`, which should supply the GA with the fitness of the member passed as a parameter. The program performs no error checking.

```
#include "extgen.h"
extern float Fitness(POPULATION *pop, unsigned member);

#define POPSIZE     100
#define CHROMSIZE   512
#define PRESSURE    1.5
#define PCROSS      0.8
#define PINV        0.6
#define PMUT       0.005


void main(void)
{  POPULATION *pop;     /* contains the main population  */
   POPULATION *newpop;  /* used to hold crossover result */
   unsigned p1, p2;     /* the chosen parent strings     */
   unsigned i;          /* loop counter                  */

   pop = DefinePopulation(POPSIZE, 1, CHROMSIZE, TRUE);
   newpop = DefinePopulation(1, 1, CHROMSIZE, FALSE);

/* The following loop creates new strings and replaces them
   into the population. When the minimal fitness wanted is
   reached by the best string in the population, the loop
   ends.                                                  */

   for(i=0; pop->member[0].fitness < MINFITNESS; i++)
   {
      p1 = RankSelect(pop, PRESSURE);
      p2 = RankSelect(pop, PRESSURE);

      Crossover(pop, p1, p2, newpop, 0, PCROSS, 2);
      Invert(newpop, 0, PINV);
      Mutate(newpop, 0, 2.0, PMUT);

      newpop->member[0].fitness = Fitness(newpop, 0);

      SortPopulation(pop);
      RankReplace(pop, newpop, 0);
   }

   SavePopulation(pop, "sample.pop", i);
}
```

When the program has finished, the last population is saved to disk (named `sample.pop`) and contains the number of iterations needed to reach the desired minimum fitness.

New genetic functions

In order to experiment with pressure values larger than 2.0, a new `RankSelect()` function was written:

```
unsigned BitRankSelect(POPULATION *pop,
                       double pressure);
```

It accepts all pressure values larger than 1.0 and returns the index of the selected member using the function:

$$\text{popsize}\frac{(1-r)}{\left(\left(\frac{p+1}{2}\right)^2\right)^r}$$

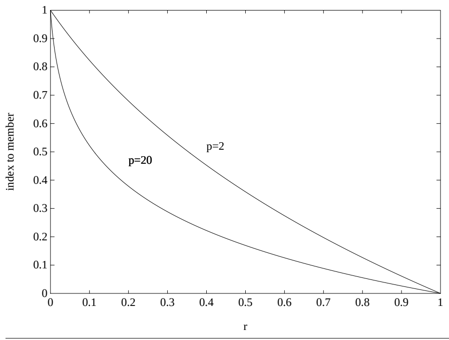


Figure 2. *The function used with BitRankSelect().*

with *r* a random number in the range [0.0,1.0) and *p* the pressure. The function is plotted for values *p*=2.0 and *p*=20.0 in figure 1. With this function, the probability to be selected for the best member in the population is *p* times as high as the median member.

```
int BitCrossover(POPULATION *oldpop,
                 unsigned parent1, unsigned parent2,
                 POPULATION *newpop, unsigned child,
                 double pCross, unsigned points);
```

The original `CrossOver()` function only supported a maximum of one crossover point per gene. In our research we used just one (long) gene, but wanted more than one crossover point. So a new crossover function was implemented that chooses the crossover points uniformly random over the whole chromosome. A new member is created as normal by concatenating the alternate parts from two parents (after the genes were temporarily put in the same order, see chapter 3).

```
void BitInvert(POPULATION *p, unsigned m, double pInv);
```

The `Invert()` function from the library swaps genes, as described in chapter 3. In our research we used just one gene, containing the (variable-length) production rules end on end. In order to have an automatic reordering operator like `Invert()`, a new function was implemented that acts on the first (and only) gene like it was a chromosome consisting of 1-bit long genes. So inversion points are chosen within the gene and the bits in between are swapped bitwise.

Chromosome to grammar

The first sub-program (`CHR2GRAM`) reads the bitstring from a member of the population and translates it into a set of production rules. This is done using the table presented in chapter 6. In order to ensure that the production rules adhere to the restrictions mentioned in chapter 6, ‘clean-up’ functions are called for each production rule. Below is an extract of the function that checks the context. Code not shown checks the context on irregularities concerning digits and commas.

```
int CheckContext(char *context)
{  unsigned i=0,
            l=0, r=0;  /* count the number of [’s and ]’s  */

   while(context[i])   /* context string terminate with \0 */
      if(context[i]=’[’)
      {  l++; i++;
      }
      else if(context[i]=’]’)
      {  if(++r > l || !i)         /* remove extraneous ] */
         {  RemoveCharacters(context,i,1);
            r--; }
         else if(context[i-1]==’[’)   /* remove empty []’s */
         {  RemoveCharacters(context,i-1,2);
            i--; }
         else if(context[i-1]==’,’)    /* remove useless , */
         {  RemoveCharacters(context,i-1,1);
            r--; i--; }
         else
            i++;
      }
   return l==r;
}
```

The functions for checking the predecessor and successor are similar.

L-system

The file containing the production rules (which also contains the axiom) is used as input for the `LSYSTEM` sub-program. This program comprises three parts: the first part reads the input and generates the string, the second part translates this string into an adjacency matrix and the third part reorganizes the matrix in order to remove unnecessary connections.
Below is an extract from the first part. It shows the function `FindProduction()`, which returns a pointer to a production data structure. The function is called for each character in the string (pointed to by the `curPtr` pointer) that is rewritten:

```
struct Production
{  char   *lCon;   /* the left context                    */
   int    lConLen; /* the length of the left context      */
   char   *pred;   /* the strict predecessor              */
   int    predLen; /* the length of the strict predecessor */
   char   *rCon;   /* the right context                   */
   int    rConLen; /* the length of the right context     */
   char   *succ;   /* the successor                       */
   int    succLen; /* the length of the successor         */
};
```

`FindProduction()` calls three other functions: `prefix()`, which returns TRUE if the current character(s) pointed to by `curPtr` are the predecessor of the current production rule and `LeftContext()` and `RightContext()`, two functions that return TRUE if the context of the current production rule is a subset of the nodes connected to the predecessor and the nodes the predecessor is connected to.

```
PRODUCTION *FindProduction(char *curPtr,
                           PRODUCTION prodSet[])
{
   while(prodSet->predLen)
   {  if( !prefix(prodSet->pred, curPtr) ||
          !LeftContext(curPtr, prodSet->predLen,
                       prodSet->lCon)    ||
          !RightContext(curPtr, prodSet->predLen,
                        prodSet->rCon))
         ++prodSet;
      else
         return prodSet;
   }
   return NULL;
}
```

In the second part the string is translated into an adjacency matrix using the `ParseModule()` function. This function connects all nodes within a module, and when a module within the current module is found, it is called recursively for that new module.

A chain of nodes (see figure 2) is not very useful with backpropagation. Also, the backpropagation library trains *all* output nodes, regardless of the number actually needed. All these extraneous nodes increase computing time and therefore the last part of the program examines the matrix and removes unused output nodes and chains of nodes.

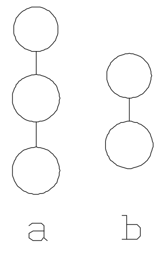


Figure 3. *The left chain can be replaced by the one on the right.*

This process (nicknamed SuperSnoei) is done using the CleanupMatrix() function. Below is an extract from this function. The adjacency matrix is contained in a two-dimensional array called adjMatrix[][]. The number of nodes in the matrix is contained in the variable nrNodes.

```
void CleanupMatrix(void)
{  unsigned i,j;
   unsigned n;       /* node counter */
   unsigned nc, nr;  /* counts for node n the number of 1’s in
                        corresponding column and row */
   unsigned from,to;

   n=0;
   do
   {  nr=nc=0;
      for(j=n+1; j < nrNodes; j++)
         if(adjMatrix[n][j])
         {  if(++nr > 1)
               break;
            to=j;
         }

      for(i=0; i < n; i++)
         if(adjMatrix[i][n])
         {  if(++nc > 1)
               break;
            from=i;
         }
      if(nc==nr && nr<=1)   /* Chain or unconnected node */
      {  for(j=n+1; j < nrNodes; j++)   /* Move one left */
            for(i=0; i < nrNodes; i++)
               adjMatrix[i][j-1]=adjMatrix[i][j];
         for(i=n+1; i < nrNodes; i++)     /* Move one up */
            for(j=0; j < nrNodes; j++)
               adjMatrix[i-1][j]=adjMatrix[i][j];

         nrNodes--;                 /* We removed a node */

         /* If the node was connected two nodes, make a
            direct connection */

         if(nc && nr) adjMatrix[from][to-1]=1;

         n=0;                       /* Rescan the matrix */
      }
      else
         n++;                      /* Done with this one */
   }
   while(n < nrNodes);
}
```

Backpropagation

The matrix that results from the LSYSTEM program is used as input for the backpropagation algorithm. The backpropagation algorithm was written in C++, because an object oriented approach lent itself admirably to the implementation of modular backpropagation. The class network implements the complete network, it makes use of the classes module and connection.

```
class network       // Implements a backpropagation network
{
   unsigned inputSize, outputSize,
            totalModules,
            inputModules, outputModules,
            totalConnections;
   class connection** connection;
   class module** module;

public:
   network(unsigned, unsigned, // size of in and output
           unsigned*,          // size of each module
           conSpec*);          // specifies the connections
   network(char, char*);       // load network, MATRIX or NET
  ~network();

   int  save(char*);    // saves the network
   void reset();        // selects 'optimal' random weights
   void train(double*,  // input
              double*,  // desired output
              double*); // calculated output, before training
   void calc(double*, double*);               // input, output
   class module* getModule(unsigned);
   class connection* getConnection(unsigned,  // fromID
                                   unsigned); // toID
   void setAlphaBeta(double,double);
   unsigned getInputSize();
   unsigned getOutputSize();
};
```

The simplest way to implement a backpropagation network is to make two arrays, one specifying the size of each module, the other specifying the connectivity. The example network of figure 10 in chapter 2 can be declared as follows:

```
unsigned mod[]={4,3,2,2,2,3,0};
conSpec con[]={{0,1},{0,2},{1,3},{2,4},{3,5},{4,5},{0,0}}

class network net(4,3,mod,con); // First two parameters are
                                // input and output size
```

Note that both arrays end with zeros to indicate the last element. A network `net` can be trained with:

```
net.train(input,output,calc);
```

that takes the input and desired output as first two parameters, the third parameter returns the output of the network itself for the presented input. The desired output needs to be calculated before learning in order for backpropagation to compute the errors (see chapter 2).

Trained networks can be saved with:

```
net.save("net.net");
```

which saves the complete current status of the network, including all weights and moment terms. Saved networks can be loaded using:

```
new class network net(NET,"net.net");
```

witch dynamically creates the network. Another way, used mostly in this research, is to load the specification for a network in matrix-form. This is done with the same function:

```
new class network net(MATRIX,"net.mat");
```

We used the convention of specifying matrix files with the extension `.mat` and network files with `.net`. A matrix file (which is in fact an adjacency matrix for the nodes of the network) that implements the same network as the last example is shown in figure 3 (note that for this example each module can be seen as a block of ones).

It is also possible to work directly with modules and connections, this enables the user to set several values (of weights etc.) himself. This was for example used to make figure 6 of chapter 2, where the saved network of figure 5 was loaded and tested after each change of the two specified weights.

```
#nodes 16
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 4. *Adjacency matrix for the example network.*

The following C++ code shows an extract of a program implementing the XOR problem:

```
// Learn XOR

#include "backprop.h"

#define ALPHA   0.4    // learning rate parameter
#define BETA    0.9    // momentum term parameter
#define MAXITER 10000  // maximum of training iterations
#define EPSILON 0.2    // error below which training is stopped

void main(void)
{  unsigned  mod[] = { 2,2,1,0 };
   conSpec con[]   = { {0,1},{0,2},{1,2},{0,0} };
   double error,
          input[2], output[1], calc[1];
   int i,n;

   class network net(2, 1, mod, con);

   net.setAlphaBeta(ALPHA, BETA);
   net.reset();
   for(n=0; n < MAXITER; n++)
   {
      error=0.0;
      for(i=0; i < 4; i++)
      {
         input[0]=(double)(i/2); // Calculate input/output pair
         input[1]=(double)(i%2);
         output[0]= (i==1 || i==2) ? 0.9 : 0.1;
         net.train(input, output, calc);
         error += output[0]-calc[0];
      }
      if(error < EPSILON)  // Stop after sufficient convergence
         break;
   }
   net.save("xor.net");
}
```

The software for backpropagation is available through the library `BACKPROP.LIB`, which can be linked to the rest of the program. Only the file `backprop.h` has to be included, which contains function prototypes, data structure definitions and predefined constants. The library can also be used with normal C programs, if they are compiled with a C++ compiler.

Main program

Two variations of the main program were written: one implementing the roulette wheel selection described by Goldberg [GOLD89], and one implementing rank based selection and one-at-a-time replacement, as described by Whitley [WHIT89A].

Both programs first read a simulation file, which contains all the necessary parameters. This file is an ASCII file containing lines starting with the # symbol, followed by a keyword. Parameters are separated by spaces and follow the keyword. An example simulation file, containing all the valid keywords is shown below.

```
## Sample simulation file
##
#files      c:\simulation\
#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    ABC
```

Empty lines and lines starting with ## are ignored (usable as comment lines). The first three keywords indicate the location of the files used during the simulation and the names of both a control file and the population file. `#size` specifies number of members in the population. `#pmut, #pcross, #sites, #pinv` and `#pressure` influence the genetic operators. `#steps` (maximum number of rewriting steps) and `#axiom` are used by the L-system.

Multiple computers can run the same simulation simultaneously. If roulette wheel selection is used, each running program reads a specified number of un-processed members from the population file and ‘locks’ them using a control file containing an indicator for each member which can be `FREE`, `PROCESSING` or `DONE`. If one of the programs notices that all indicators are `DONE`, the population file is locked and the fitness values in the population file are updated and the control file is reset to `FREE`. Also the fittest member found in this generation is archived and a new population is created.

If one-at-a-time replacement is used, each program opens the main population file and creates a small local population file containing a specified number of newly created strings. Then each program processes these local members and when all have been assigned a fitness, the program locks the main population file and replaces its local members into the main population using `RankSelect()`. It then fills its local population file with new strings and starts processing again.

In both cases, the processing is done by presenting a member to CHR2GRAM, which writes its output to a file. The name of that file is then transferred to LSYSTEM, whose matrix output file is finally passed to BACKPROP.

Because the population-update loop of the one-at-a-time version very much resembles the example given at the beginning of the chapter, only an extract from the roulette wheel selection version is given.

```
if((p=DefinePopulation(size, 1, CHROMSIZE, FALSE))==NULL)
   ErrorExit(NO_MEMORY, __FILE__, __LINE__);

if((np=DefinePopulation(3, 1, CHROMSIZE, FALSE))==NULL)
   ErrorExit(NO_MEMORY, __FILE__, __LINE__);

if((file=fopen(popFile, "rb+"))==NULL)
   ErrorExit(READ_ERROR, __FILE__, __LINE__);

if(LoadPopulation(p, file, &g)!=GEN_NO_ERROR)
   ErrorExit(READ_ERROR, __FILE__, __LINE__);

for(i=0; i < p->popSize; i++)
{
    do
    {  p1 = Select(p);
       p2 = Select(p);
    }
    while(p1 == p2);  /* the two parents must be different */

    CopyMember(p, p1, np, 0);  /* copy to temp. population */
    CopyMember(p, p2, np, 1);

    BitInvert(np, 0, pInv);
    BitInvert(np, 1, pInv);

    BitCrossover(np, 0, 1, np, 2, pCross, sites);

    Mutate(np, 2, 10, pMut);

    if(SaveMember(&np->member[2], i, file, 1, CHROMSIZE) !=
          GEN_NO_ERROR)
       ErrorExit(WRITE_ERROR, __FILE__, __LINE__);
}

fclose(file);
```

The results of the experiments that were done with the software can be found in chapter 8. A number of suggestions to improve and/or expand the software can be found in chapter 9.

'Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work.'
— Thomas A. Edison

8 Experiments

This chapter presents some of the results from the experiments done throughout the research. The following problems were tried using the developed software:

- exclusive OR (XOR)
- TC problem
- handwritten digit recognition
- 'where' and 'what' categorization
- mapping of $[0.0,1.0)^2$ values onto four categories

In the following paragraphs, each of the problems is described and some of the results are given.

Exclusive OR (XOR)

The XOR function (see also chapter 2) is a boolean (logical) function of two variables:

```
f(0,0) = 0
f(0,1) = 1
f(1,0) = 1
f(1,1) = 0
```

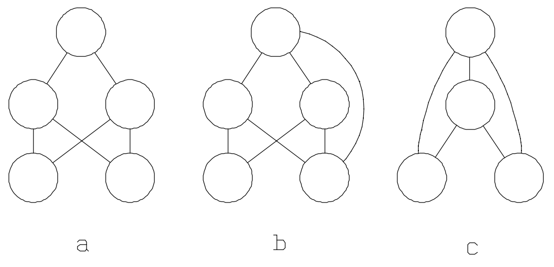


Figure 1. *Three sample XOR networks.*

It was proven in 1969 that a network able to solve this problem should have a hidden layer [MINS69]. Some of the 'standard' solutions of networks able to solve XOR are shown in figure 1. In this experiment we tried both an L-system where the comma is used to separate two neighbouring nodes/modules, as well as an L-system where a minus is used to make a specific connection between neighbouring nodes/modules (see chapter 6). Both methods worked fine, but the method using minus symbols was faster in finding modules (groups of internally unconnected nodes) whereas the comma method 'preferred' chains of nodes. Because of this, we decided to use the minus symbols for the rest of this research instead of the comma symbols.

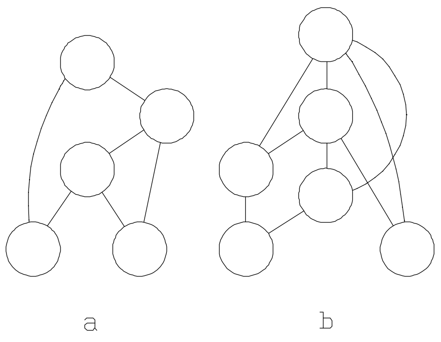


Figure 2. *Two other networks for XOR.*

All three networks shown in figure 1 were found. Two of the other networks that were able to learn XOR and were found are shown in figure 2. The matrix cleanup method described in chapter 7 ('SuperSnoei') was not used in this experiment and the networks shown are the result of 'human' pruning of the resulting adjacency matrices.

TC problem

With the TC problem, a neural network should be able to recognize the letters T and C in a 4x4 grid (see also [RUME86]). Each letter, consisting of 3x3 pixels, can be rotated 0, 90, 180 or 270° and can be anywhere on the 4x4 grid. The total number of input patterns is therefore 32 (there are 4 positions to put the 3x3 grid). Figure 3 shows the eight possible 3x3 patterns and a sample 4x4 grid that would be presented as input to the network. A black pixel was represented with an input value of 0.9, and white pixels with an input value of 0.1. The output node was trained to respond with 0.1 for a T and with 0.9 for a C. An approximation of the maximum possible error we used to calculate the fitness of the network is:

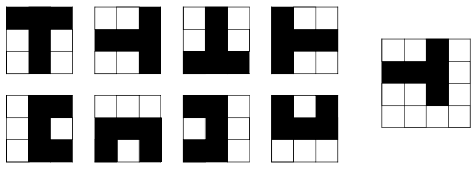


Figure 3. *The 8 possible letters and one sample input grid of 4x4.*

$$error = \sqrt{0.9^2 \cdot 32} \approx 5.1$$

and the fitness was calculated as 6 - *error*.

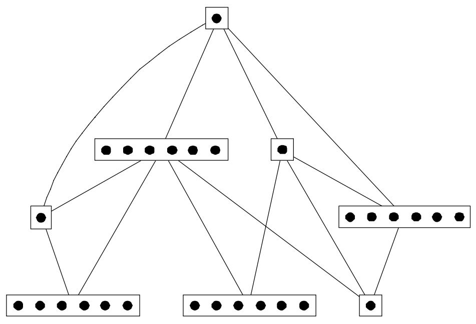


Figure 4. *The network found for the TC problem.*

A network with 16 input nodes would be expected (each of the pixels of the 4x4 grid is presented to one input node), but the network that was found (shown in figure 4) has only 13 input nodes. Even with three input nodes less then expected, this network was able to learn all input combinations (the fitness of the member containing this network was 5.9, after roughly 7500 recombinations).

We also compared this network to a number of simple backpropagation networks with one hidden layer. We varied the number of nodes in the hidden layer for these standard networks from 3 to 8. We presented the 32 patterns 250 times to each network. Then each network was tested 50 times. The results are shown in figure 5. Each bar represents the 50 test cycles and the dark (lower) part equals the number of times the network did not classify *all* 32 pattern correctly. As can be seen in figure 5, the found network outperformed all one hidden layer networks easily. For the networks with one hidden layer, the optimum is a hidden layer of 6 nodes.

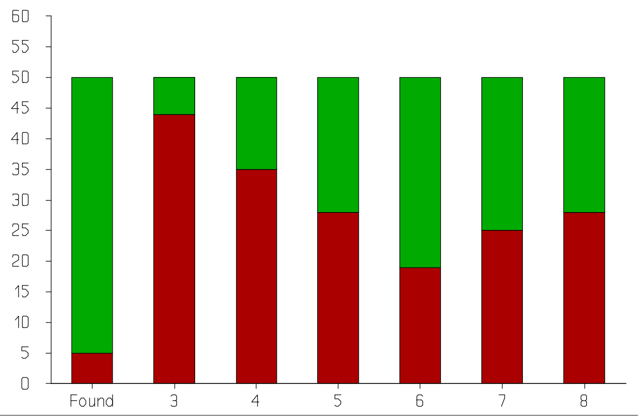


Figure 5. *Comparison between found network and standard networks.*

Handwritten digit recognition[1]

For this problem, a neural network should be found that is able to recognize handwritten digits 0,1..,9, presented on a 5x5 grid. 200 handwritten digit stimuli were obtained by instructing 20 human subjects to write the digits 0,1..,9 over a 5x5 grid using a computer mouse. The proportion blackness in each square of the grid constituted an activation value. The network is trained with one half of the stimuli, the other half was used to test network generalization capabilities. The data was originally used as a test for network design principles using genetic algorithms and CALM networks [HAPP92] (for a short introduction to CALM, see inset). One of the networks found during that test is shown in figure 6. This particular network had an correct generalization score of 81%.

It proved to be too simple a problem for backpropagation because our method found a network *without* a hidden layer (25 input and 10 output nodes) that classified 97% of the test set correctly

The CALM model (Categorizing And Learning Module) [MURR92] was proposed as a candidate for the incorporation of modular information processing principles in artificial neural networks. CALM is a modular network algorithm that has been especially developed as a functional building block for large modular neural networks. A number of neurophysiological constraints are implemented in CALM by using the general architecture of the neocortical *minicolumns* [SZEN75] as a basic design principle. These modules contain about 100 neurons, and an important feature of these modules is the division in *excitatory* pyramidal cells, which form long ranging connections to other cortical regions, and various types of short range, within module, *inhibitory* interneurons. The structural principle of *intramodular inhibition* implies that the main process within a module will be a *winner take all competition* [PHAF91],[MURR92]. Competition enables a system to autonomously categorize patterns and, therefore, can be used to implement unsupervised competitive learning. Categorization and learning have also been incorporated in the CALM model. Basically a CALM module consists of a number of *representation-nodes* (R-nodes) that are fully connected to the input through learning connections. The number of R-nodes in a module is referred to as its *size*. Categorization in a CALM module is operationalized as the association of an input pattern with a unique R-node that is said to *represent* the pattern. Different patterns are categorized in CALM by repeated subsequent presentations of a set of input vectors. During and following the categorization process learning takes place, which preserves the association between input pattern and R-node by adjusting the learning weights. The categorization proceeds through the resolution of *competition* between all R-nodes that are activated by the pattern, mediated by specialized inhibitory veto-nodes (V-nodes). The winning R-node constitutes the representation of the pattern. Different activation patterns will lead to different representations. The specific learning rule of CALM enables it to even discriminate non-orthogonal patterns.

Short introduction to CALM. (For a more complete description, the reader is referred to [MURR92]).

[1] *For this experiment training and test data from [HAPP92] were used.*

(and 100% of the training set). However, one of the advantages of the CALM network is the ability to learn *new* patterns without interfering too much with the earlier learned patterns.

‘Where’ and ‘what’ categorization

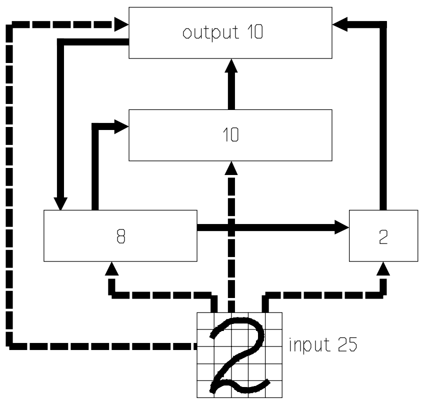


Figure 6. *CALM network used for handwritten digit recognition.*

Another problem we tried was proposed by Rueckl et al. [Ruec89] where, like the TC problem, a number of 3x3 patterns had to be recognized on a larger grid. With this problem, there are 9 patterns (see figure 7), which are placed on a 5x5 grid. Besides recognizing the *form* of the pattern, the network should also encode the *place* of the pattern on the larger input grid (of which there are 9 possibilities). Rueckl et al. conducted these experiments in an attempt to explain why in the natural visual system *what* and *where* are processed by separate cortical structures (e.g. [Livi88]). They trained a number of different networks with 25 input and 18 output nodes, and one hidden layer of 18 nodes. The 18 output nodes were separated in two groups of 9: one group for encoding the form, one group for encoding the place. It appeared that the network learned faster and made less mistakes when the hidden layer was split and appropriate *separate processing resources* were dedicated to the processing of *what* and *where* (see figure 8). Of importance was the number of nodes allocated to the *form* and *place* system respectively. Figure 8 shows the optimal network found, where 4 nodes are dedicated to the processing of *place* and the remaining 14 nodes to the more complex task of processing *form.* Analysis of these results by Rueckl et al. revealed that the processing of *what* and *where* strongly interfere in the un-split model (one hidden layer of 18 nodes).

When we tried this problem using our method, the optimal network found was again a network *without* a hidden layer. Although this seems surprising, it is not difficult to see that a network without a hidden layer is very capable of learning the problem. The specific split network found by Rueckl et al. is only necessary if a hidden layer is used: only then interference can occur.

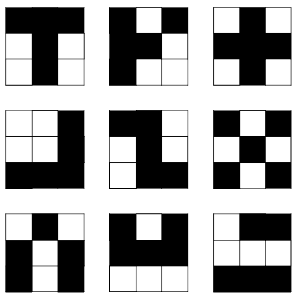


Figure 7. *The nine patterns.*

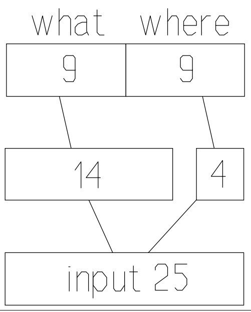


Figure 8. *The network from [Rueck89].*

Mapping problem

The last problem tried was more of a problem for standard backpropagation networks with one or no hidden layer. The original problem was one of the experiments done by Van Hoogstraten [Hoog91] in order investigate the influence of the structure of the network upon its ability to map functions. In the experiment, he created a two-dimensional classification problem with an input space of $[0,1)^2$. From this space 100 (10 x 10) points (*x*,*y*) are assigned to four classes (he used colours, we use the symbols ■ ● ▲ ✶). He constructed two mappings, where the second was derived from the first by ‘misclassifying’ three of the 100 points. The second mapping is shown in figure 9. The misclassified points are (0.4,0.4), (0.5,0.0) and (0.7,0.6) and can be seen as *noise*. Although Van Hoogstraten wanted networks that were not fouled by that noise (and therefore ignored them), we were interested in networks that are able to learn *all* points correctly.

Van Hoogstraten tried a number of networks, all of which had two input nodes (for *x* and *y* respectively) and four output nodes, one for each symbol. Both a network with a hidden layer of 6 nodes as a network with a hidden layer of 15 nodes were more or less able to learn the first mapping (without the noise), but failed to learn the three changed points of the other mapping. Another network with a hidden layer of 100 nodes was able to learn one of the misclassified points, but not all of them. Only when he used a network with three hidden layers of 20 nodes each (with 920 connections!), all three misclassified points were learned correctly. We tried this experiment hoping our method would find a *small*, *modular* network that was able to learn the second mapping correctly. For this experiment the following values for the various parameters were used:

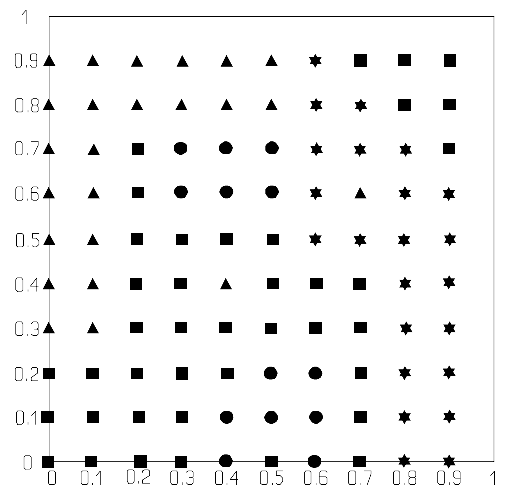


Figure 9. *The input grid for the mapping problem.*

- population size 500, chromosomes of length 1024
- Whitley selection and replacement, pressure of 1.4
- mutation probability 0.01
- crossover probability 0.5, 10 crossover points
- inversion probability 0.7
- L-system axiom A, with a maximum of 6 rewriting steps

It took three days and 11 Sun Sparc4 workstations to converge to the last network shown in figure 10. It was found after roughly 35,000 string evaluations. The other networks shown in figure 10 can be seen as its 'ancestors' and were found earlier in the simulation.

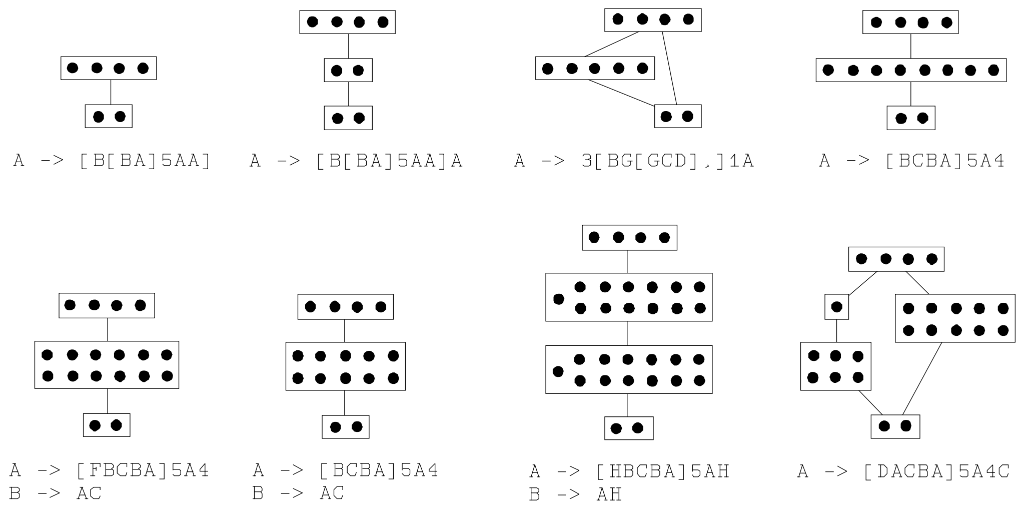
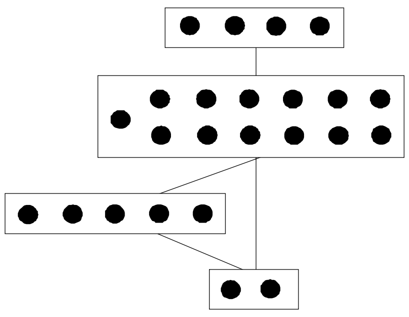


Figure 10. *The networks found during the simulation.*

For each of the networks the production rules used to create the network are also shown. Clearly, the production rules evolved from each other.

Figure 12 shows the fitness of the worst and best member of the population throughout the simulation, as well as the average fitness value. The fitness was determined by adding the total error of the output for each of the input stimuli, which was subtracted from 2000. So a low error resulted in a high fitness.

It took three more days to reach the network shown in figure 11, which was found after roughly 85,000 string evaluations.



```
A -> [BCBA]5A4
B -> AC
A > CC ->
```

Figure 11. The last network found.

In comparison to the 2-20-20-20-4 network used by Van Hoogstraten, the last network from figure 10 and the network from figure 11 had a consistently higher fitness and took less time to train because the network contains only 146 connections instead of the 920 connections of the other network. After extended training, the network from figure 11 had an error of 14, versus 74 for the 2-20-20-20-4 network. Figure 13 shows the convergence of the networks from Van Hoogstraten vs the genetically found network shown in figure 11, during one training session.

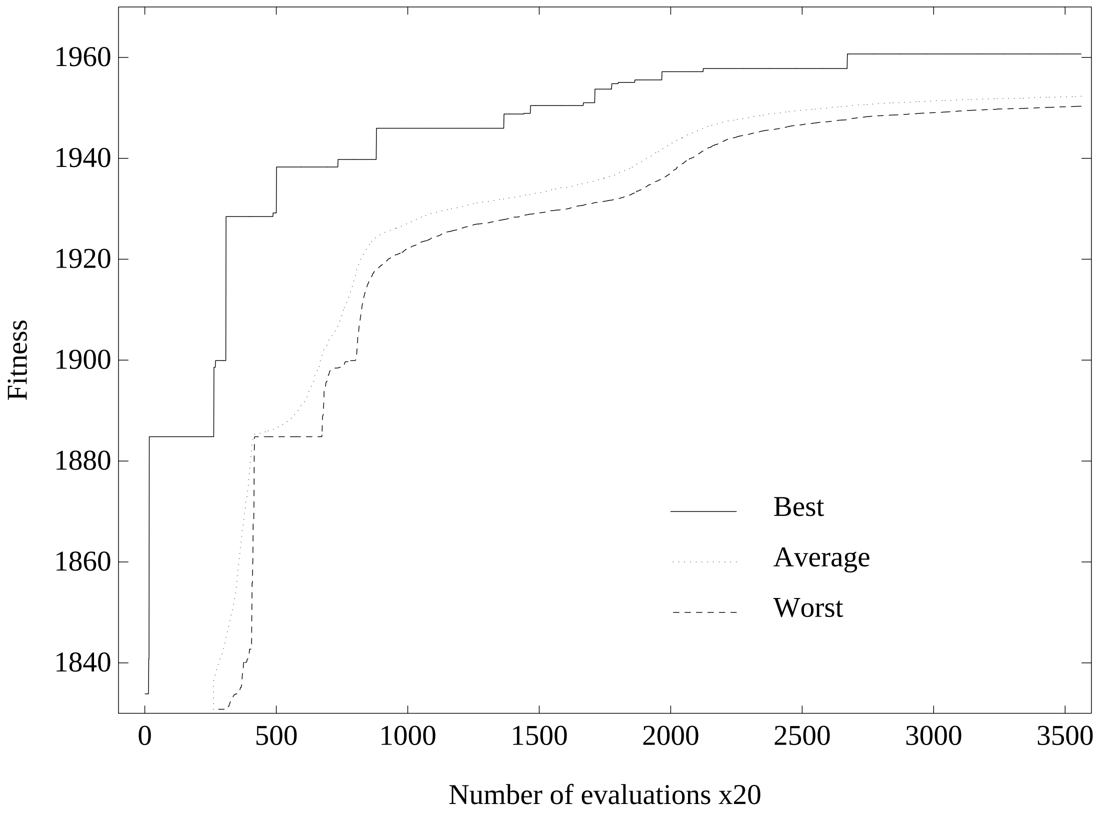


Figure 12. Graph showing the fitness of the population during one simulation.

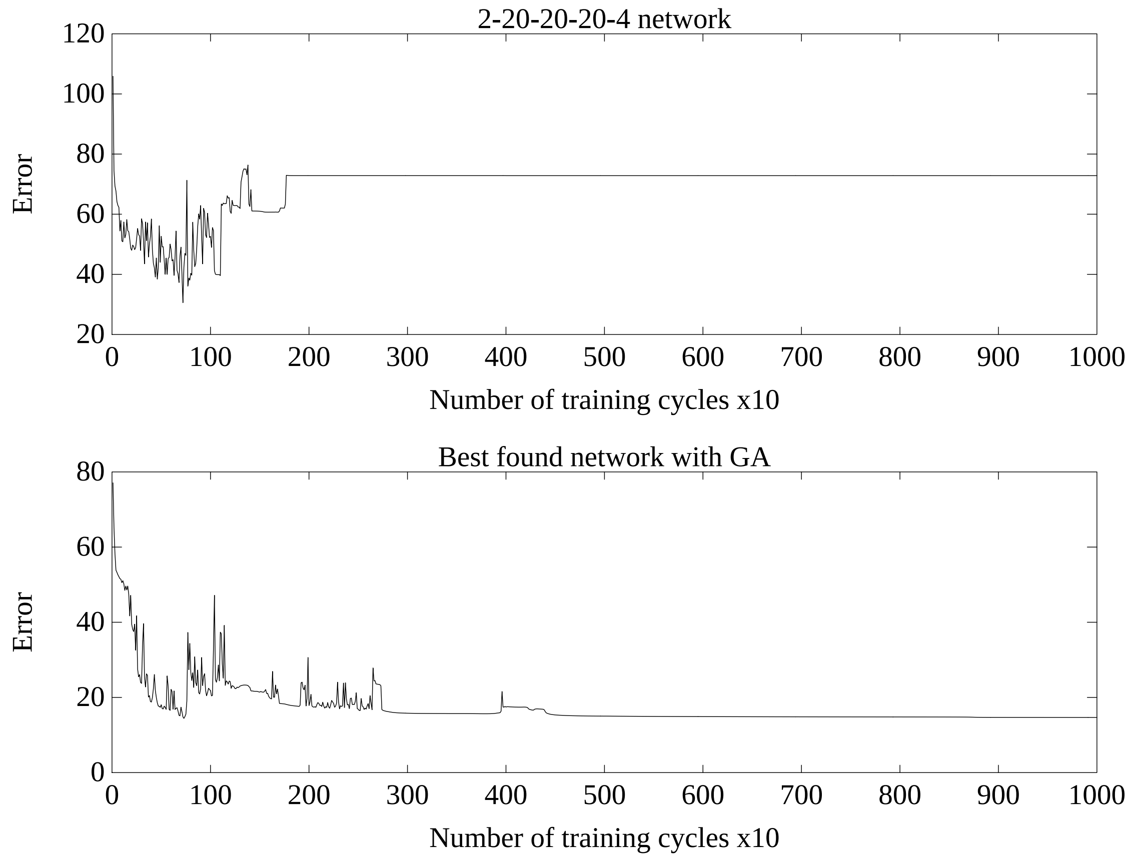


Figure 13. *Comparison of convergence.*

‘Perfection is attained by slow degrees;
it requires the hand of time.’
— Voltaire

9 Conclusions and recommendations

Although in a very early stage, the combination of search methods, growth mechanisms and computational principles based on nature and biology results in a very promising strategy for designing artificial intelligent systems, no matter what definition one chooses for artificial intelligence. One of the major problems in the future will be the general acceptance of the products based on these principles. The systems will become very complex, and their functioning will no longer be understandable. It is very difficult for people to accept a conclusion from an artificial intelligent system, not knowing how the system came to its result, even if the results of the system are just as good, or even better than results of human experts. This is something we have to get used to in the future. Perhaps continuing research of these *complex systems* will result in new mathematical techniques that *are* able to explain what is happening inside a working neural network.

Conclusions

Genetic algorithms are being used to find architectures for neural networks that are best suited for a specific task. With a genetic blueprint representation, good results can be obtained for small networks. This blueprint method does not work well for large size networks, because of the exponential growing number of possible connections that have to be coded. The goal of this research was to design a method that allowed, in principle, all possible network architectures, but did not suffer from the exponential growth of the bits needed to represent each member of the population. The method that resulted from this research uses the metaphor of a recipe, where not the network itself is coded, but just a set of rules to produce that network. Preliminary results, shown in chapter 8, suggest that the method does work, and finds modular networks that perform better than networks without specific structure. The idea to use rules of a grammar that *directs* the genetic search towards modular architectures seems to be justified with the results that were obtained. However, a lot of experimenting still needs to be done in order to optimize the proposed method.

One of the problems encountered was the amount of computing power needed. The mapping problem described in chapter 8 needed about a week to converge to its final solution, using 11 Sun Sparc4 workstations in parallel. This is caused by the large amount of computing needed to evaluate each architecture.

Further research

During this research just one branch out of many has been tried. As usual, after working intensively on a project, large numbers of unanswered questions remain and a lot of ideas could not be implemented and tested in six months, the duration of this project. In the remainder of this chapter, a number of these ideas are presented as well as some suggestions for improvements and possible further research. Each suggestion is followed by a short explanation. Also, some parallels with nature are pointed out.

Backpropagation and other network paradigms

As already mentioned, it takes a lot of computing time to train a network, which has to be repeated for each new chromosome during a simulation. However, a lot of networks that are generated, especially when the genetic algorithm is more or less converged, have the same topology.

Make a database containing networks already evaluated. If a network is evaluated, say, ten times and the average error is used to calculate the fitness of that network architecture, the chance of a chromosome coding for that network receiving an erroneous fitness is small. If a population is almost converged, and is generating many equal network structures, it may be a good idea to create a database containing already trained network architectures. This will reduce the amount of computing time needed considerably. This only works if the genetic algorithm generates a discrete output. As soon as floating point parameters are coded, the probability of two members being equal becomes too small.

Prune extraneous input nodes. With the SuperSnoei pruning method, only extraneous output nodes and chain ares removed. However, extraneous input nodes can also be removed from the network.

The genetic algorithm

Calculate the fitness of the initial population. While using the algorithm based on Whitley [WHIT89A], we initialized the population with random chromosomes. The fitness of all randomly determined initial members were however not evaluated, and were set to zero. This might have caused a direct loss of useful information that happened to be present in the random initialization. When each initial member is tested the moment it is placed in the population, the selection of parents begins with correct fitness values.

Initialize the population with members having a non zero fitness. The previous suggestion can be improved by making sure each initial member *has* a fitness. In this research it happened very often that a newly generated chromosome had no

> fitness (a fitness of zero). This could happen if the network that resulted from this string had less output nodes than a given minimum, or was simply too large, given a maximum number of nodes. It also happened frequently that a string did not contain any production rules that could be applied at all. When the program was running, these new chromosomes were of course not inserted in the population. But the selection can already start at the initialization of the population. Only insert the next initial member if it has a non-zero fitness. This will probably also result in a more diverse population, the probability that several initial members are inserted with the same production rules is very small. It may also prevent an evolution that gets stuck with one accidental reasonable set of production rules, which can not evolve any further. This can very well be seen as a local optimum in evolution.

This last idea gives rise to a comparison with the '*primeval soup*' theories (see e.g. [ORGE73]). The atmosphere of the earth at the time when there was no life on earth, contained no oxygen but plenty of hydrogen and water, carbon dioxide, and very likely some ammonia, methane and other simple organic gases. The 'primeval soup' theory states that this environment, under influence of lightning and ultraviolet light (there was no ozone layer), after thousands of millions of years spontaneously created some molecules that were able to replicate themselves, and subsequently started evolution. Once started, evolution could slowly give rise to more and more complex creatures. This is something that can be found in our simulations as well. It sometimes took quite a while to find a production rule that could be applied, but after that, an increasing number of useful new chromosomes were found. The experiments that were described in chapter 8 can, according to us, be compared only to the period just after the start of evolution, given the small part of the full potential of our method that was actually used. Much longer simulations on much larger tasks may perhaps require the full potential of our method. The last suggestion will keep the 'primeval soup' period as short as possible, and will start the simulation with a high amount of randomly found rules that, when combined, may quickly result in good solutions.

> **Make an initial population with 'useful' production rules.** The previous suggestion may become substantially faster if the initial population already contains several sets of production rules that can create known standard solutions of the problem at hand. Also, production rules found at previous simulations may be taken. It needs to be tested whether this method is really faster than waiting for random members, because the evolution from the initial handmade population to members with the wanted solution, may well be a longer process than starting from scratch. This is particularly likely to happen when the optimal solution is something completely different from the structure that was expected.

> **Include the axiom in the genetic process.** In our simulations we used axioms that did not change during the simulation. They were fixed in a parameter file that was used by the genetic algorithm. Including the axiom as a separate gene in the members of the population will probably find better axioms than our ad hoc chosen axioms. Some production rules that are very useful may only come into existence given specific axioms on which they operate.

This last suggestion also leads to a link to biology, because the genetic information that is stored in the nucleus of the cell, can only be read (the process of transcription) with a special enzyme (RNA polymerase) that is already present in the cell. The human ovum can be seen as the axiom on which the production rules coded in the genes can build, and contains all that is necessary for the initial embryo to start growing.

Use chromosomes with a different size. During our simulations we worked mostly with chromosomes of 1024 bits. But looking at the results, we noticed that most solutions were made using just a few production rules. By decreasing the size of the chromosomes, the amount of freedom is reduced which may result in a faster convergence of the genetic algorithm. This however, may also have its negative effects, because the chance of finding a production rule in the initial population that can be applied to the axiom will be much smaller using small chromosomes.

Use chromosomes with a variable length. Another idea that could be tried is to have the genetic algorithm decide for itself what the most useful chromosome length will be. This needs a change of the genetic algorithm. Crossover has to be changed in such a way that the parts of both parents that are copied into the new child are chosen so that the child has the preferred chromosome size. This size can be coded as a separate gene, coding the length of the individual members of the population.

This idea too has its counterpart in nature where each different species has its own number of chromosomes. Slight deviations in the number or size of the chromosomes in nature usually lead to functional disorders of the organism, like Down's syndrome with humans (mongolism). But as can be safely assumed, some changes have to lead to improvements.

In reality, the individuals of a population never have an eternal life. This, we believe, is not an accident, and is probably a necessary condition for evolution to work in the real world. During our many tests on different network structures for the same problem we noticed very often that accidental good initial weights could result in a reasonably good performing trained network. See for example figure 5 in chapter 8. Even the worst network with a hidden layer of three nodes found a correct input/output mapping several times. But it clearly is not the wanted solution, because we were looking for network structures specific for this problem that *always*, or at least as many times as possible, correctly learned the task. Now suppose this network is generated in an early stage of a simulation, and receives a high fitness because of an accidental good initialization of its weights. It will stay in the population and, because of its high fitness, will be chosen as a parent more frequently than other members of the population. If that situation continues for a long time, there will be offspring of this bad network that, again accidentally, is able to learn the task. This will result in an increasing part of the population containing copies of the original bad member. Now suppose that the production rules of this bad architecture are such that small changes will lead to even worse network architectures. This means that the genetic algorithm is trapped in a local optimum caused by the randomness that is present in backpropagation. But it is just this kind of randomness that we were trying to eliminate by finding the right architecture. During the test phase of our software we have indeed sometimes run simulations that ended in local optima. That is why we trained each generated network more than once in later simulations, each time using another random initialization of the network's initial weights, taking the average error of those separate training as a measure for the fitness of that architecture.

Use 'aging'. An idea that came up when discussing the above problems was to 'age' the population. The artificial aging of the members of a static population can be done by multiplying all fitness values of the members with a value slightly below one, just after every new chromosome that is evaluated. Of course this only concerns the static population model we used (see chapter 3), because the variant described by Goldberg generates a completely new population every generation. The result of this will, hopefully, be that the unwanted members eventually disappear from the population because their offspring will only by chance have a high enough fitness to be inserted into the population. If a network is found that is ‘really’ good, which means that the network continuously generates offspring with high fitness, the original good network will, as a consequence of the aging process, slowly leave the population, but its children will carry its information on to next generations. It is important not to choose the *aging factor* (the percentage that is taken from the fitness each time) too large, because a chromosome with a high fitness should stay in the population long enough to generate enough offspring. Also, a copy of the best member found so far should be saved, just in case.

Re-evaluate each member. A variation on the last suggestion is to regularly re-evaluate each member of the population (again only with a static population model), for example each time a number of new chromosomes equal to the population size is generated. Members that have a low probability of repeatedly scoring a high fitness will now also leave the population.

These suggestions are necessary when genetic algorithms with a static population are used in areas where the fitness has a large random component. Current research on genetically optimizing CALM networks (see inset chapter 8), at the group where this research took place, already is using the idea of aging. They are constructing a CALM network that has to be able to learn sequences in time, and they had major problems with the noise used in the CALM modules, because it occasionally resulted in a correct sequence. This resulted in an occasional high fitness, and resulted in a population with a very high level of noise.

The L-system

The suggestion about axioms above, also applies to this section.

Use wild cards in the production rules. These wild cards can be used to improve and extend production rule matching in the context. For example, a lower case letter can be used to match *any* digit (and the same letter can be used in the predecessor to copy that digit). Also empty brackets (`[]`) can be used to indicate *any* module.

Use a different number of rewriting steps. In all experiments done throughout this research, the number of rewriting steps for the L-systems was six. Only if the string reached a maximum length, less rewriting steps were used. Different numbers of rewriting steps should be examined to see what effect it has on the resulting networks.

The above idea can be extended by not using a fixed number of rewriting steps, but by varying this number during the rewriting.

Do not use a fixed number of rewriting steps. For example, the L-system can continue rewriting until either a maximum string length is reached, or the network created sofar has sufficient input and output nodes. Also, after each rewriting step, the network created sofar could be tested and the fitness values should be stored. When a predetermined maximum string length or number of rewriting steps is reached, the highest fitness so far could be returned.

The production rules, their coding and the resulting strings

Use another translation table. Although the table chosen for the translation from chromosome to characters in a production rule (see figure 11, chapter 6) was based on the genetic code, the actual distribution of the characters in the table was chosen rather arbitrarily. Experiments should be done with different distributions: not only the placements of the characters in the table, also the number of each of the characters can be varied (more asterisks, less digits, etc.). Also more symbols can be introduced by increasing the size of the table, for example to allow larger skips.

If other network learning paradigms are used, where the feedforward restriction does not apply, provisions have to be made in order for the strings being able to represent recurrent networks.

Use negative skips. These negative skips (which can be represented by special symbols, or maybe the minus symbol can be used with the skips currently used) are needed to allow for recurrent networks. This suggestion also implies changes to the translation table.

Examine the number of building blocks. In this research, it is assumed that by using overlapping information in the chromosomes, the number of building blocks evaluated with each string evaluation is higher than with standard genetic algorithms that do not use overlapping strings. It should be examined how large the amount of building blocks being processed with each string evaluation actually is.

The above suggestion can be extended to strings with different density.

Use less overlapping strings. The amount of building blocks resulting from the above suggestion should be compared to the amount of building blocks for strings with less overlapping information. For example, the strings could only be read forward (resulting in a string being read 6 times). As a result of these experiments, the amount of overlapping information could be changed.

Try a fourth type of string. One type of strings we thought about, but have not tried, are strings were *each* letter represents a complete module. The size of these modules can be fixed (for example, A has size 1, B has size 2, etc.), or a separate gene can be used to code the length of each module. One advantage of this method could be that modular networks can be represented with shorter strings. However, if the number of production rules needed (and their length) does not change, this method may not have any advantages.

Miscellaneous

A large simulation should be tried. The experiments and simulations done throughout this research were all of a small scale. In order to investigate the real potential of the methods proposed, a large simulation should be done. Preferably, because of the amount of computing time needed, a supercomputer or large network of (for example) transputers should be used, so adjustments to the used method can be easily made, without having to wait several days for a simulation to finish.

More exhaustive comparisons should be made. The method proposed should be compared more exhaustively with other methods, like the ones using blueprints. The methods should be compared according to convergence of the GA, the performance of the networks found, the different length of chromosomes used etcetera.

Investigate scalability. If an appropriate problem is found, the possibilities of scalability of the method proposed should be investigated. Convergence of the GA and performance of the networks found should be compared with the same problem at larger scales, as well as with other methods.

Make a tool to display network architectures. When the networks get larger and larger it will be more and more difficult to get an impression of the structure of the networks. A tool for drawing networks from adjacency matrices will be necessary.

A Derivation of backpropagation

This appendix contains the derivation of the backpropagation learning algorithm (the *generalized delta rule* GDR). The explanation is restricted to a minimum, the interested reader can find a more extensive treatment in [RUME86], or in [FREE91]. The derivation will be given for a simple three-layer network, but can easily be generalized for networks with more layers. Several other artificial neural network classes are treated in [FREE91], which offers a good general introduction into the field of neural networks.

The error for an output node during training is $\delta_j = (y_j - o_j)$, with y_j the desired output, and o_j the actual output for the jth output node. The error that is minimized by the GDR is:

$$E = \frac{1}{2}\sum_{j=1}^{r} \delta_j^2 ,$$

with r the number of output nodes. To determine the change of weights to the output nodes, we calculate the negative gradient $-\nabla E$ with respect to the weights w_{ij}^o. Considering each component of ∇E separately we get:

$$E = \frac{1}{2}\sum_{j=1}^{r} (y_j - o_j)^2$$

and

$$\frac{\partial E}{\partial w_{ij}^o} = -(y_j - o_j)\frac{\partial \mathrm{f}}{\partial (stim_j^o)}\frac{\partial (stim_j^o)}{\partial w_{ij}^o}$$

where

$$\frac{\partial\left(stim_j^o\right)}{\partial w_{ij}^o} = \frac{\partial}{\partial w_{ij}^o}\sum_{i=1}^{q} w_{ij}^o h_i + \theta_j = h_i ,$$

with q the number of hidden nodes, h_i the ith hidden node, and therefore:

$$-\frac{\partial E}{\partial w_{ij}^o} = \left(y_j - o_j\right) \mathrm{f}'\left(stim_j^o\right) h_i .$$

The weights are changed proportional to the negative gradient, so:

$$w_{ij}^o(t+1) = w_{ij}^o(t) + \Delta w_{ij}^o(t) ,$$

where

$$\Delta w_{ij}^o = \alpha\left(y_j - o_j\right) \mathrm{f}'\left(stim_j^o\right) h_i .$$

The factor α is the *learning-rate parameter*. Notice the requirement that the function f be differentiable. The function that is usually used is:

$$\mathrm{f}(stim) = \left(1 + \mathrm{e}^{-stim}\right)^{-1} .$$

The derivative of f is:

$$\mathrm{f}' = \mathrm{f}(1-\mathrm{f}) = o_j\left(1 - o_j\right) ,$$

so

$$w_{ij}^o(t+1) = w_{ij}^o(t) + \alpha\left(y_j - o_j\right) o_j\left(1 - o_j\right) h_i .$$

This is the formula used to update the weights from the hidden layer to the output layer. If

$$\delta_j^o = \left(y_j - o_j\right) \mathrm{f}'\left(stim_j^o\right) = \delta_j \mathrm{f}'\left(stim_j^o\right) ,$$

we can rewrite the formula as:

$$w_{ij}^o(t+1) = w_{ij}^o(t) + \alpha \delta_j^o h_i . \qquad (*)$$

The problem with the calculation of the weights from the input layer to the hidden layer is that the desired activation for each node in the hidden layer is, contrary to the output layer, unknown. But the total error E is related to the activation of the nodes in the hidden layer:

$$E = \frac{1}{2}\sum_{j=1}^{r} \left(y_j - o_j\right)^2$$

$$= \frac{1}{2}\sum_j \left(y_j - \mathrm{f}\left(stim_j^o\right)\right)^2$$

$$= \frac{1}{2}\sum_j \left(y_j - \mathrm{f}\left(\sum_{i=1}^{q} w_{ij}^o h_i + \theta_j\right)\right)^2 .$$

It is now possible to calculate the gradient of E with respect to the weights from the input layer to the output layer:

$$\frac{\partial E}{\partial w_{ij}^h} = \frac{1}{2}\sum_{k=1}^{r} \frac{\partial}{\partial w_{ij}^h}(y_k - o_k)^2$$

$$= -\sum_k (y_k - o_k)\frac{\partial o_k}{\partial\left(stim_k^o\right)} \frac{\partial\left(stim_k^o\right)}{\partial h_j} \frac{\partial h_j}{\partial\left(stim_j^h\right)} \frac{\partial\left(stim_j^h\right)}{\partial w_{ij}^h}$$

$$= -\sum_k (y_k - o_k)\,\mathrm{f}'\left(stim_k^o\right) w_{jk}^o\,\mathrm{f}'\left(stim_j^h\right) x_i .$$

with x_i the *i*th input node. The weights to the hidden layer are updated with the negative of this gradient:

$$\Delta w_{ij}^h = \alpha \mathrm{f}'\left(stim_j^h\right) x_i \sum_{k=1}^{r} (y_k - o_k)\,\mathrm{f}'\left(stim_k^o\right) w_{jk}^o$$

or

$$\Delta w_{ij}^h = \alpha \mathrm{f}'\left(stim_j^h\right) x_i \sum_k \delta_k^o w_{jk}^o .$$

Notice that every weight update on the weights to the hidden layer depends on *all* the error terms δ_k^o of the output layer. This is where the notion of *backpropagation* arises. The known errors on the output layer are *propagated back* to the hidden layer to determine the appropriate weight changes on that layer. By defining

$$\delta_j^h = \mathrm{f}'\left(stim_j^h\right)\sum_{k=1}^{r} \delta_k^o w_{jk}^o$$

we can write the weight update formula for the weights to the hidden layer:

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

which has the same form as (*). Both have the same form as the *delta rule*, which is used for training networks without hidden layer (see e.g. [FREE91]).

B Entropy of neural networks

All throughout this study we claimed and showed examples of the fact that the initial structure given to a neural network greatly determines its performance. This appendix gives a more mathematical foundation to these claims. It is partly based on [SOLL89].

Learning as entropy reduction

A neural network, when trained, is performing an *input-output mapping*. The mapping that is implemented by the network depends on the architecture of the network and its weights. When the architecture is fixed, the mapping of network is solely determined by the weights. The set of all possible weight configurations (the weight space) of a network determines a *probability distribution* over the space of possible input-output mappings that can be implemented with the fixed architecture. The *entropy* of this distribution is a quantitative measure of the diversity of the mappings realizable by the architecture under consideration.

Learning from examples reduces the intrinsic entropy of the untrained network by excluding configurations which realize mappings incompatible with the training set. The residual entropy of the trained network measures its generalization ability. The goal of this research has been to design a method to find a network structure for a given task, that has a residual entropy of zero after training from examples, which means that the network converges to a state that implements the desired mapping independent of the initial weight configuration. If a network architecture is able to give several different mappings as a result of the training from examples, the residual entropy will not be zero. This will probably lead to a bad generalization because only *examples* of the complete domain are used for training, and it is possible to extract the same set of examples from many different mappings. A correct topology of the network will only allow for the mapping that is actually wanted. If the residual entropy after training is not zero, small deviations from the input patterns in the training set can give large errors in the output. This can be seen by imagining a weight space with a large number of minima, each implementing exactly the mapping of the training set, but just one (or none) representing the actual wanted mapping. It also shows the importance of the selection of the training set.

Definition of entropy

Before it is possible to give a definition of the entropy of a neural network, the probability of a network implementing a mapping f has to be defined. Given a network architecture and its corresponding weight space W (the set of all possible weight settings w), Sara Solla [SOLL89] defines the *a priori probability* of the network for a mapping f as

$$P_f^0 = \Omega_f / \Omega_W,$$

where Ω_W equals the total volume of the allowed weight space, and

$$\Omega_f = \int_{\Omega_W} \Theta_f(w) \mathrm{d}w$$

is the volume in the initial weight space that implements the desired mapping with

$$\Theta_f(w) = \begin{cases} 1 & \text{if } w \text{ implements mapping } f \\ 0 & \text{otherwise} \end{cases}$$

So in other words: P_f^0 is the probability the network implements the desired mapping through an arbitrary setting of w. Obviously the network should be able to implement the desired mapping: $P_f^0 \neq 0$. If $P_f^0 = 0$, the network is unable to learn the desired input/output mapping.

Selecting a network structure defines a class of functions that are realizable with that network. A useful measure of the diversity of possible mappings f that can be implemented with the chosen architecture is the entropy [DENK87]

$$S^0 = -\sum_{\{f\}} P_f^0 \ln\left(P_f^0\right)$$

of the a priory probability distribution. Since $P_f^0 \neq 0$ not just for the desired mapping f but also for mappings $f' \neq f$, the distribution of P_f^0 is such that $S^0 > 0$.

The definitions of the *a priory probability* and the *entropy* of neural networks give a mathematical notion of the necessity to find networks with a good initial structure. A correct initial structure of a network results in a high a priory probability, which in turn leads to a low intrinsic entropy of the untrained network. It is the intrinsic entropy S^0 of the untrained network that needs to be eliminated via a learning process. The purpose of training is to confine the configuration space to the region $\{w \mid \Theta_f(w) = 1\}$, thus eliminating all ambiguity about the input-output mapping implemented by the trained output.

Addendum

At the time this thesis was almost finished, we found a reference to an article by Kitano [KITA90] with the title: "Designing neural networks using genetic algorithms with graph generation system". This addendum gives a summary of the method used by Kitano and highlights the main differences between the method he proposed and the one proposed in this thesis.

Kitano uses a standard genetic algorithm to search for artificial neural networks. The GA is used in combination with a kind of graph grammar which can also be called a *matrix rewriting system*. With this grammar, each production rule has a single symbol (*non-terminal*) as left-hand-side, and the right-hand-side consist of a 2x2 matrix, either containing non-terminals, or 1s and 0s. The axiom is also a 2x2 matrix. For each rewriting step, the non-terminals are replaced by the 2x2 matrix from the matching production rule. 1s and 0s are replaced by a 2x2 matrix of all 1s or all 0s respectively. Note that the matrix grows by a factor two in size with each rewriting step. This process can be repeated a number of times. If any non-terminals remain after the last rewriting step, they are replaced by 0s. The binary matrix that results from the last rewriting step is used as an adjacency matrix for the network to be created. A (very) small example is shown in figure 1.

With the production rules:

```
A → 10   B → 11   C → 01
    11       10       01
```

The axiom: can be rewritten as:

```
AB                 1011
CA                 1110
                   0110
                   0111
```

Figure 1. *Sample production rules.*

The chromosomes used by Kitano for the genetic algorithm are fixed length binary strings consisting of two parts: in the first part, on which genetic operators such as crossover and mutation are performed, for each of the non-terminals, a production rule is coded that rewrites that symbol into other non-terminals. In the second part, which is not changed during the genetic search, for 16 predetermined non-terminals a matrix consisting of 1s and 0s is pre-encoded. For a more precise description of the production rules used, as well as the rewriting method itself, the reader is referred to [KITA90].

With the method proposed by Kitano, the final size of the network is determined *directly by the number of rewriting steps*. Although his method has the advantage that with the same chromosome length different size networks can be coded and that for larger networks no exponentially larger chromosomes are needed, the method proposed in this thesis has even less constraints on the resulting network size (although restraints *can* be made, if necessary).

Another important difference between his method and the one proposed here, is the fact that Kitano's method does not use any *context*. We believe that, since context plays an important role in biological growth, there should be context incorporated in any method that is supposed to be biological plausible (and that is exactly what Kitano tried to do).

The results from [KITA90] do indicate his method works and that the method has less scaling problems than can be expected with so-called *blueprint* methods. However, to our knowledge, the method proposed by Kitano has not been used by many others. Perhaps he was too early with his proposals, perhaps his method was not flexible enough (for example because no context was used).

References

[ALLP80] D.A. Allport; 'Pattern and actions'. In: *New directions in cognitive psychology*, G.L. Clagton (Ed.), Routledge and Kegan Paul, London, 1980.

[CREU77] O.D. Creutzfeldt; 'Generality of the functional structure of the neocortex'. In: *Naturwissenschaften 64*, 507-517, 1977.

[DAWK86] R. Dawkins; *The blind watchmaker*, Longman, 1986. Reprinted with appendix by Penguin, London, 1991.

[DENK87] J.S. Denker, D.B. Schwartz, B.S. Wittner, S.A. Solla, R.E. Howard, L.D. Jackel and J.J. Hopfield; 'Large automatic learning, rule extraction and generalization'. In: *Complex systems, 1*, 877-922, 1987.

[DODD90] N. Dodd, 'Optimization of network structure using genetic algorithms'. In: *Proceedings of the International Neural Network Conference, INNC-90-Paris*, 693-696, B. Widrow and B. Angeniol (Eds.), Kluwer, Dordrecht, 1990.

[FREE91] J.A. Freeman and D.M. Skapura; *Neural networks: algorithms, applications and programming techniques*. Addison-Wesley, Reading, 1991.

[GARI90] 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.

[GAZZ89] M.S. Gazzaniga; 'Organization of the human brain'. In: *Science, 245*, 947-952.

[GOLD89] D.E. Goldberg; *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, Reading, 1989.

[GUYT86] A.C. Guyton; *Textbook of medical physiology*. Saunders, Philadelphia, 1986.

[HAPP92] B.L.M. Happel; *Architecture and function of neural networks: designing modular architectures*. In prep., 1992.

[HARP89] S.A. Harp, T. Samad and A. Guha; ‘Toward the genetic synthesis of neural networks’. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their applications (ICGA)*, 360-369, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

[HECH90] R. Hecht-Nielsen; *Neurocomputing*. Addison-Wesley, Reading, 1990.

[HEEM91] J.N.H. Heemskerk and J.M.J. Murre; ‘Neurocomputers: parallelle machines voor neurale netwerken’. In: *Informatie, 33-6*, 365-464, 1991.

[HOFS79] D.R. Hofstadter; *Gödel, Escher, Bach: an eternal golden braid*. Basic Books, New York, 1979.

[HOGE74] P. Hogeweg and B. Hesper; ‘A model study on biomorphological description’. In: *Pattern Recognition, 6*, 165-179, 1974.

[HOLL68] J.H. Holland; *Hierarchical descriptions of universal spaces and adaptive systems*. University of Michigan Press, Ann Harbor, 1968.

[HOLL75] J.H. Holland; *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Harbor, 1975.

[HOOG91] R.J.W. van Hoogstraten; *A neural network for genetic facies recognition*. Unpublished student report, Leiden, 1991.

[HUBE62] 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.

[JERI85] H.J. Jerison; ‘Issues in brain evolution’. In: *Oxford surveys in evolutionary biology, 2*, 102-134, R. Dawkins and M. Ridley (Eds.), 1985.

[KAND85] E.R. Kandel and J.H. Schwartz; *Principles of neuroscience*. Elsevier, New York.

[KITA90] H. Kitano; ‘Designing neural networks using genetic algorithms with graph generation system’. In: *Complex Systems, 4*, 461-476, Champaign, IL, 1990.

[KOCH05] H. von Koch; ‘Une méthode géometrique élémentaire pour l’étude de certaines questions de la théorie des courbes planes’. In: *Acta mathematica, 30*, 1905.

[LIND68] A. Lindenmayer; ‘Mathematical models for cellular interaction in development, parts I and II’. In: *Journal of theoretical biology, 18*, 280-315, 1968.

[LIVI88] M. Livingstone and D. Hubel; ‘Segregation of form, color, movement and depth: anatomy, physiology and perception’. In: *Science, 240*, 740-749, 1988.

[MAND82] B.B. Mandelbrot; *The fractal geometry of nature*. Freeman, San Francisco, 1982.

[MARI90] B. Maricic and Z. Nikolow; ‘GENNET - Systems for computer aided neural network design using genetic algorithms’. In: *Proceedings of the International Joint Conference on Neural Networks, Washington DC, 1*, 102-105, 1990.

[MARS74] J.C. Marshall and F.J. Newcombe; *Journal of physiology, 2*, 175, 1974.

[MINS69] M. Minsky and S. Papert; *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[MOUN75] V.B. Mountcastle; ‘An organizing principle for cerebral function: the unit module and the distributed system’. In: *The mindful brain*, G.M. Edelman, V.B. Mountcastle (Eds.), MIT Press, Cambridge, MA, 1975.

[MURR92] J.M.J. Murre; *Categorization and learning in neural networks. Modelling and implementation in a modular framework.* Dissertation, Leiden University, 1992.

[ORGE73] L.E. Orgel; *The origins of life*. Wiley, New York, 1973

[PARK85] D.B. Parker; *Learning logic*. MIT Press, Cambridge, MA, 1985.

[PENR89] R. Penrose; *The emperor's new mind*. Oxford University Press, New York, 1989.

[PHAF91] R.H. Phaf; *Learning in natural and connectionist systems: experiments and a model.* Unpublished dissertation, Leiden University, Leiden, 1991.

[POSN72] M.I. Posner, J. Lewis, C. Conrad; In: *Language by ear and by eye,* 159-192, J.F. Kavanaugh and I.G. Mattingly (Eds.), MIT Press, Cambridge, MA, 1972.

[POSN86] M.I. Posner; *Chronometric explorations of mind*. Oxford University Press, 1986.

[POSN88] M.I. Posner, S.E. Peterson, P.T. Fox and M.E. Raichle; 'Localization of cognitive operations in the human brain'. In: *Science, 240*, 1627-1631, 1988.

[PRUS89] P. Prusinkiewicz and J. Hanan; *Lindenmayer systems, fractals and plants*. Springer-Verlag, New York, 1989.

[PRUS90] P. Prunsikiewicz and A. Lindenmayer; *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.

[RUEC89] 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.

[RUME86] D.E. Rumelhart and J.L McClelland (Eds.); *Parallel distributed processing. Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.

[SCHW88] J.T. Schwartz; 'The new connectionism: developing relationships between neuroscience and artificial intelligence', 1988.

[SOLL89] S.A. Solla; 'Learning and generalization in layered neural networks: the contiguity problem'. In: *Neural networks: from models to applications*, 168-177, L. Personnas and G. Dreyfus (Eds.), I.D.S.E.T, Paris, 1989.

[SZEN75] J. Szentagothai; 'The neural network of the cerebral cortex: a functional interpretation'. In: *Proceedings of the Royal Society of London, B, 201,* 219-248, 1975.

[SZEN77] J. Szentagothai; 'The 'module-concept' in the cerebral cortex architecture'. In: *Brain Research, 95,* 475-496, 1977.

[SZIL79] A.L. Szilard and R.E. Quinton; 'An interpretation for D0L-systems by computer graphics'. In: *The Science Terrapin, 4,* 8-13, 1979.

[TURI63] A. Turing; 'Computing machinery and intelligence'. In: *Computers and thought,* E.A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.

[WARR82] E.K. Warrington; The fractionation of arithmetical skills: a single case study. In: *Quarterly journal of experimental psychology: human experimental psychology, 34, A(1),* 31-51, 1982.

[WERB74] P.J. Werbos; *Beyond regression: new tools for prediction and analysis in the behavioral sciences*. Unpublished Ph.D. thesis, Harvard University, Cambridge, MA, 1974.

[WHIT89A] 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),* 116-121, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

[WHIT89B] D. Whitley and T. Hanson; ‘Towards the genetic synthesis of neural networks’. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their applications (ICGA)*, 391-396, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

[ZEKI88] S. Zeki and S. Shipp; ‘The functional logic of cortical connections’. In: *Nature*, *335*, 311-317, 1988.