Rijksuniversiteit te Leiden

Vakgroep Informatica

**Local Structure Optimization
in Evolutionary Generated
Neural Network Architectures**

M.V. Borst

MASTER'S THESIS

Department of Computer Science
Leiden University
P.O. Box 9512
2300 RA Leiden
The Netherlands

SEIS

GA*i*N

Local Structure Optimization in Evolutionary Generated Neural Network Architectures

M.V. Borst

Preface

This thesis is the result of research done at the department of Computer Science at Leiden University in the Netherlands. It is an extension of the work done by Egbert Boers and Herman Kuiper [Boers92].

The research incorporated several parts. First, extensive reading was done to get familiar with the areas of genetic algorithms, neural networks and algorithms that change the structure of neural networks. Second the developed software by Egbert Boers and Herman Kuiper was modified so that it could use the CMU data benchmark (developed at the Carnegie Mellon University at Pittsburgh, USA) and a few tests with the Cascade Correlation Learning Architecture algorithm of Scott Fahlman and Christian Lebiere [Fahlman90] were done. Than some criteria to detect if a module in a modular neural network lacks computational power were developed together with four methods to install the newly added complexity. All this was implemented an tested on various problems and the results are presented in this thesis

I would like to thank Egbert Boers for his patience, understanding and help with the construction of both the ideas behind the programs as the programs themselves. Also I would like to thank Ida Sprinkhuizen-Kuyper for her trust, enthusiasm and guidance throughout the project. Furthermore a lot of credit is due to Roberto Lambooy who has helped me, with certain paragraphs and my rotten English. Finally I would like to thank Scott Fahlman for maintaining the CMU Benchmark Collection.

Leiden, September 1994

Marko Borst

Abstract

This thesis is an extension of the work done by Boers and Kuiper [Boers92]. In their master thesis they proposed a method to produce good *modular* artificial neural network structures. It was argued that *modular* artificial neural networks have a better performance than non-modular networks. Based on the natural process that resulted in our brain, they introduced a *genetic algorithm* to imitate evolution and used *L-systems* to model the kind of *recipes* nature uses in biological growth.

In this research the objective was to find a local optimization method for *modular* neural network structures. Such a method should change the structure of the network. Since adding an additional unit to an already existing module was shown to be the most 'local' way of changing the structure of a network a couple of criteria was developed to decide if and which module of the network needs an additional unit. Besides that, four, on constructive algorithms inspired, methods to install the new unit in the module were tested. The results of the methods on various test problems indicate that if a additional unit is added on bases of its amount of incoming weight changes, this generally results in better network structures. The method of installing a new unit in a modular neural network based on the method of installing a new unit in Cascade Correlation (a *constructive* neural network algorithm developed by Lebiere and Fahlman [Fahlman88]) seems best.

When a local optimization method is added to an genetic algorithm it may speed up the genetic search process with a considerable margin [Hinton87], this is called the *Baldwin effect* [Baldwin1896]. Preliminary results with the combination of the local neural network structure optimization method and the algorithm of Boers and Kuiper to generate neural network structures seem to agree with this effect, but a lot more simulations have to be done before on this respect a definite conclusion can be drawn.

Contents

1   Introduction

1.1   Research Goals

One of the biggest problems in the use of neural networks nowadays is the problem of finding an appropriate structure for a given task. An ideal structure is a structure that independently of the starting weights of the net, always learns the task, i.e. makes almost no error on the training set and generalizes well. Boers and Kuiper [Boers92] produced an algorithm to find a 'good' *modular* neural network structure to solve a given task. It used a genetic algorithm to produce a grammar (based on L-systems) that itself was used to produce modular networks. In this research I tried to find some local structure optimization methods for modular neural networks. Such a method can, when used in combination with a genetic modular neural network generator like the one of Boers and Kuiper, speed up the search for 'good' structures.

After a short explanation of Neural Networks and their problems (chapter 2) the advantages of *modular* back-propagation will be explained. Then some algorithms that modify the structure of Neural Networks will be discussed (chapter 3). Besides a short overview of the various methods, two inspiring methods will be explained: The Cascade Correlation Learning Architecture of Fahlman and Lebiere and The Growing Cell Structures algorithm of Fritzke [Fritzke93]. Chapter 4 starts with a short explanation of *Genetic Algorithms* and of L-systems. Further it describes how learning can guide evolution, commonly known as the *Baldwin effect* [Hinton87]. In the next chapter a few restrictions on local structure optimization methods will be described. For one, true local optimization method, i.e. adding an extra unit to an already existing module, a few possible criteria will be discussed. These criteria are used to decide if a module needs an extra unit. Further a few ways to initialize the newly created weights will be shown. Chapter 6 describes a few implementation issues, so that it will (hopefully) be easier to use the program. In chapter 7 the results of the various criteria and installing methods on a few tests will be shown, together with the results of some larger problems. In chapter 8 some conclusions will be presented along

with ideas for further research. In the rest of this chapter all the main ideas behind this research are presented briefly.

1.2   Neural Networks

A computer is just a machine, it can be used to simplify or perform certain tasks. But what kind of tasks? Well obviously all tasks that need complex arithmetic operations and tasks that require large amounts of data storage. Over the past decades researchers have tried to develop computer programs that where capable of performing complex tasks. Some tasks are so complex that the program has to be very sophisticated, i.e. it has to be an 'intelligent' program.

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, back-propagation, 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.

1.3   Algorithms that modify the structure

One of the major problems with neural networks is that it is very hard to know beforehand the size and the structure of a neural network one needs to solve a given problem. The obvious solution is to use the computer for this task. After a brief introduction of two different types of algorithms that try to construct a network *during* training, two of these algorithms will be presented in chapter 3.

1.4   Genetic Algorithms

Another way to find good topologies for neural networks for a given problem is to use a genetic algorithm. *Genetic Algorithms* are based on Darwin's evolution theory [Darwin1859]. Using Darwin's three principles: of variation, of heredity and of selection, Genetic Algorithms implement an evolutionary process. Samples from a problem space to be optimized are put together in a population and are subjected to so-called genetic operators: selection, crossover and mutation, reproduction, forming successive generations. The quality, called fitness, measured in terms of the problem to be optimized will approximate the best solution possible.

1.5   L-systems

In nature the precise form of a species is not coded in its genes. Instead the genetic information of a species is more a kind of *recipe* [Dawkins86]. Since researchers had already used a kind of *reversed engineering* to come up with the idea to create artificial neural networks to obtain 'intelligent' behaviour with a computer program, Boers and Kuiper looked for a way to code neural networks structures with 'recipes' [Boers92]. They used a complex form of an *L-system*. L-systems were introduced by Lindenmayer to model the growth process of plants [Linden68]. One can think of an L-system as a kind of grammar. The biggest difference with 'standard' grammars is that all characters in a string are *rewritten* in *parallel*. In §4.1 a brief explanation of L-systems is presented along with the variant Boers and Kuiper [Boers92] used. For a more thorough treatment of L-systems see Prusinkiewicz and Lindenmayer [Prunsik90]. For an explanation of the variant used by Boers and Kuiper and its transformation to artificial neural networks see [Boers92].

1.6   Learning and Evolution

After the theory that learned behaviour by a species was coded back into its genes (so called Lamarckian evolution) was rejected by (most of) the scientific community in favour of the evolution theory by Darwin [Darwin1859], the attention for the influence of learning on evolution became low. Baldwin [Baldwin1896] suggested that learning could speed up the process of evolution even though the learned behaviour was not coded back to the genes. Hinton and Nowlan [Hinton87] showed that this so called *Baldwin effect* could also speed up (artificial) Genetic Algorithms. The Baldwin effect is further explained in §4.3.

2   Neural Networks

Some people find it incredible what computers nowadays can do. They are impressed by the millions of instructions per second, the giga bytes of storage capacity and the complex task these machines perform. Maybe some of this 'respect' is caused by the fact that computers are good at things, where most humans are not specifically good at: complicated computations, processing large amounts of data, etc. So, are computers more intelligent than humans? Well that depends, of course, on how you define and how you measure intelligent behaviour. If we use the method that young children use among themselves, how far and how fast someone can count, humankind is bound to loose.

It is surprising, however, that behaviour that we do not see as particular difficult or 'intelligent' cause a lot of trouble to 'computers', i.e. to programmers who want to write traditional algorithms to solve such tasks. For example, people can see in an instant of there is an empty chair in a room, a traditional computer program, that gets input from a camera would take a lot of time to complete such a task.

Well if 'traditional' methods do not work, why not try a concept that has worked in many areas of research: the concept of *reversed engineering*. That concept can roughly be described as: look for something that works, try to understand it and then try to (re)build and use it. Since humans are normally considered as 'intelligent', it may be a good idea to try to understand how humans can perform 'intelligent' behaviour, i.e. it may be fruitful to look how a human brain works and how and if its processing principles are usable in a computer program.

2.1   The Neuron

The human brain consists of a large number of interconnected *neurons*. Each of these neurons shows rather complex bio-electrical and bio-chemical behaviour. Since these neurons are the main 'actors' in our brain, research with the goal to

construct a computer program that imitates the brain concentrated on the working of neurons to create *artificial neural networks*.

A neuron can be separated in three functional parts: *axon*, *cell body* and *dendrites* (see figure 1). The dendrites receive information (*neurotransmitters*) from other neurons and transmit those signals by electro-chemical means to the cell body. The body collects all these electrically charged substances (signals) and if
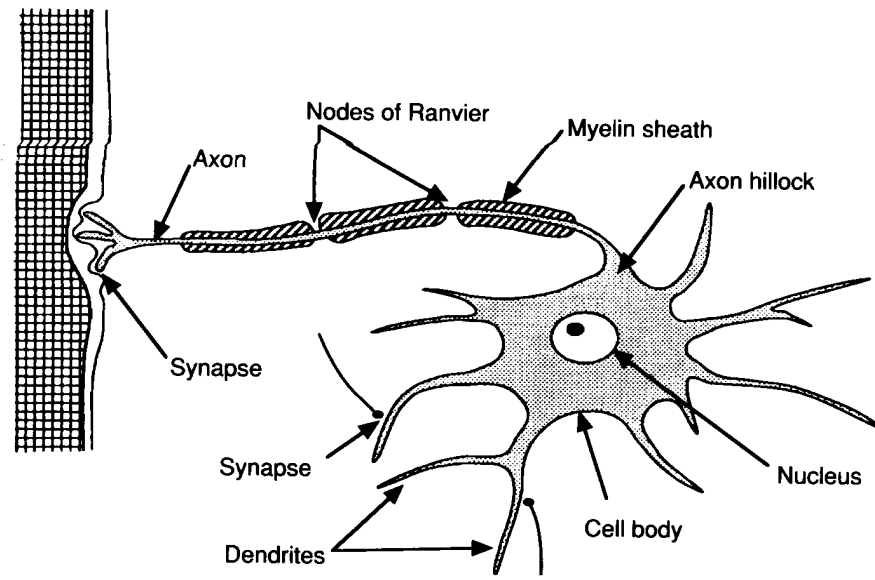
figure 1 The neuron.

the total potential of the substances in the cell exceeds a certain *threshold*, then the axon is activated, i.e. the neuron *fires*. This means that the axon transmits its activation to the dendrites of other neurons, i.e. it releases neurotransmitters. The communication from one neuron to another takes place at the *synaptic junction*, or *synapse*. The synapse is a small gap between the axon of one neuron and a dendrite of the next.

2.2   The Human brain

The brain consists of about 10^{11} neurons. If each neuron would be connected to every other neuron then our heads would have a diameter of 10 kilometres, because of the huge amount of wiring [Heemsk91]. Instead the brain is divided in several regions, most of which consists of several regions again. The smallest neuron structures consists of about 100 neurons and are called *mini-columns*. Besides this subdivision, one can also (try to) divide the brain according to *functional areas* (e.g. the visual area). Usually there are relatively few connections between areas with different functions. This strong modularity is partly suggested by patient studies, see for instance Gazzaniga [Gazzaniga89].

Despite this strong modularization the brain still has some 10^{15} connections. Even if one would know exactly how a brain works, this number alone makes a computer program that simulates a brain practically impossible at present time.

2.3   The Artificial Neuron

So it is (still) impossible to build an artificial brain that imitates the natural brain in all its detail. Some (very) large simplifications are necessary to be able to construct artificial neural networks that are capable of 'learning' some interesting 'functions'.

Where the neurons in the brain get charged chemicals in their cell bodies, artificial neurons get real numbers. Not only the activation of a neuron n determines how much neurotransmitters the dendrite of another neuron m gets, but also the 'strength' of the synapse of n to m. To model this, the real number an artificial neuron j gets from an another artificial neuron i is the product $w_{ij}x_i$, x_i is the activation of the input neuron i, and w_{ij} is called the *weight* of the connection between neuron i and j. In nature a neuron will either stimulate (i.e. give *excitatory* signals) or destimulate (give *inhibitory* signals) another neuron to fire and although the amount of stimulation may change in time, it keeps stimulating or destimulating the other neuron. In most artificial neural networks this 'restriction' is not applied, i.e. the 'sign' of a connection between to neurons may change during time. In this research these changes are permitted. It is the property of neurons that allow them to change the 'strength' of a connection with another neuron that is seen as the way the our brain *learns*.

The most obvious function neurons perform, is collecting their 'real-valued' inputs and determining their activation from that. If this activation exceeds a certain threshold, then the neuron will fire. In artificial neural networks (ANN) the stimulation is simply the (weighted) sum of all the inputs. In most cases a bias is added, which shifts the activation relative to the origin, to model the threshold. So the stimulation of an artificial neuron is:

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

The weights in artificial neurons are a metaphor for the amount of neurotransmitters transmitted by the synapses. The connections in artificial neural networks can be either negative or positive, as changed during the learning process. It should be noted that, in ANNs, the stimulation is not the same as the activation of the neuron:

$$act \ = \ f(stim)$$

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

Although f is determined for a large part by the type of ANN being used (see also §2.4.5), the basic functioning of neurons is globally the same, since all ANNs are in one way or another based on the original brain.
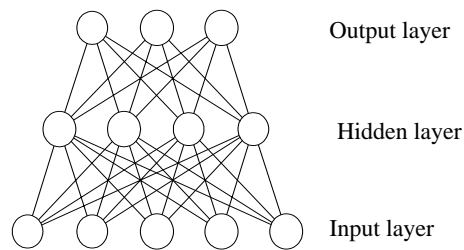
figure 2 A typical backpropagation network

2.4 Artificial Neural Networks

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

2.4.1 The Training Set

Since it is normally impossible to present a network with all possible inputs, we only present it with part of it, the training set. This set has to be chosen in such a way that the network also gives correct output for an input that was not in the training set. If the network also responds well to inputs that were not in the training set, it is said to *generalize* well. Often an ANN is trained with one set of patterns (the training set) and tested with another (the test set). If the training set was not a good representation of all possible inputs, the network probably will not perform too well on inputs that are not in the training set. Generalization is quite similar to interpolation in mathematics.

2.4.2 Backpropagation Networks

Probably the best known artificial neural network learning paradigm is backpropagation. It was first formalized by Werbos [Werbos74] and later by Parker [Parker85] and by Rumelhart and McClelland [Rumelhart86]. It it used to train a multi-layer feed forward network with supervised learning. Normally a backpropagation Network (BPN) has an input and an output layer, and a certain amount of hidden layers. The input and output layers are mandatory, the number of hidden layers is free, but often a single layer is chosen. Nodes in a certain layer only get input from other lower layers, which means that the input layer does not get any input from within the net, and only give output to nodes in higher layers, which means that nodes in the output layer do not give output to other nodes. This constitutes the feed forward principle: input only comes from lower layers and output only goes to higher layers. There is no recurrence in a feed forward network, although there are some adaptations of the BPN paradigm that allow a limited amount of recurrence. An example of a BPN with one hidden layer is shown in figure 2. The subsequent layers are fully connected.

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

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

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

$$\delta_{o,i} = o_i (1 - o_i) (y_i - o_i)$$

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

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

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

$$\Delta w_{ij}(t+1) = \alpha \delta_{o,j} h_i + \beta \Delta w_{ij}(t)$$

with $\Delta w_{ij}(t+1) = w_{ij}(t+1) - w_{ij}(t)$. All weights in the network are adapted this way, starting with the weights closest to the output layer and then working down.

This makes the overall error or *Sum sQuared Error* (SQE):

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

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

When we take an *n+1*-space, with *n* the number of weights in the network, we can plot the total error of the net for all input as a function of all the weights. This space is called the error surface. Since we want the network to perform as well as possible, we want to find the minimum in this error space. The function drawn in this space can be seen as a surface across which we let a marble roll during learning with backpropagation: it always follows the steepest gradient, or

the direction that goes down as fast as possible. However, it is possible that the error surface not only has the wanted global minimum, but also some local minima. The fact that a certain point on the error surface is a minimum, means that the surface goes up on all surrounding sides. This means that when the marble hits a minimum, it will stay there. This is of course rather unfortunate when the found minimum is not a global, but a local minimum: the network gets stuck in the local minimum.

2.4.3 Problems With Backpropagation

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

Another problem associated with backpropagation is that the place at which one starts on the error surface (which is determined by the initial weight settings, which are often random) determines whether or not a good or the best solution is found. When a solution is found that performs well on the training set, the network might still perform badly on the overall set of input, if the training set was not representative. One danger in backpropagation is for the network to get *overtrained*. This is the case when the performance of the network on the training set still increases but the performance of the net on the test set decreases. This means that the net did not look at similarities over the input, but simply learned all associations by heart. If presented with output not in the training set, the network will likely respond with other output than the desired. This problem only occurs when the network is still further trained, even though it already gives correct output. The network has learned to detect global features at first, but is trained longer and reaches such a specialization in the given training set, that it loses its ability to generalize: there is no need for generalization, it already knows every input/output pairing. This will have resulted in perfect scores on the training set. Overtraining can only happen if the network is large relative to the training set. In this case training is better stopped before full conversion on the training set is reached. Other ways to prevent overtraining is using a smaller network or adding noise to the input. Both methods will result in poorer performance on the training set, but will lead to better overall results.

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

A last problem is the occurrence of interference. This occurs when a network is supposed to learn similar tasks at the same time. Apart from the fact that smaller networks are unable to learn to many associations —they simply are full after a certain amount of learned associations— there is also the danger of input patterns being so hard to separate, that the network can't find a way to do it. When we look at the problem we can take it's input and divide that into categories. If we plot this, we would get a problem space. The network has to fill this space with figures of such a form that all inputs from the same category are included in the same figure. This means that the network has to encode in some way these forms. The more complex these forms (or the more precise they have to be) the harder it is for the network to learn it. This means that inputs that are close together, little room for a separating line, and figures with strange forms (the more concave, the worse) are hard to learn. An example of such interference between more classifications is the recognition of both position and shape of an input pattern [Rueckl89]. Rueckl 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, resulting in 81 different combinations of shape and position. The network had to encode both form and place of a presented stimulus in the output layer. It appeared that the network learned faster and made less mistakes when the tasks were processed in separated parts of the network, while the total amount of nodes stayed the same. 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 of the pattern and 14 to the apparently more complex task of the shape of the pattern. It should be emphasized that Rueckl et al. tried to explain why form and place are processed separately in the brain. The actual experiment they did, showed that processing the two tasks in one unsplit hidden layer caused interference. What they failed to describe, however, is that removing the hidden layer altogether, connecting input and output directly, leads to an even better network than the optimum they found using 20 hidden nodes in separate sub-networks as shown by Boers and Kuiper [Boers92]. The problems mentioned, however, do not occur solely with backpropagation, a lot of other network paradigms suffer from it. This brought on the search for *modularity*, which we already find in the brain.

2.4.4   Modular Backpropagation

Until now we have discussed only simple networks, where every layer is fully connected to the next. However, this is not due to a limitation in the backpropagation's learning rules. More complicated networks are created by, for instance, adding hidden layers. This doesn't really add to the computational power of the network — in fact, it has been proven that all continuous functions that can be represented by a network with more than one hidden layer can also be represented by a network with one hidden layer — although we would need an infinite number of nodes in the hidden layer for the error to approach 0, but more hidden layers do enhance the speed with which the network learns, especially for highly nonlinear inputs, inputs that are hard to differentiate.
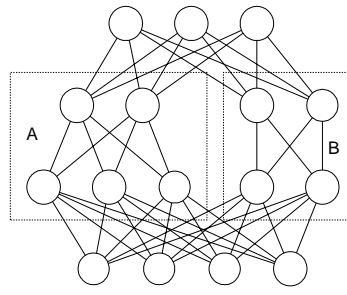
figure 3 An example network made
of two separated sub-networks

When more hidden layers are used, all layers are still fully connected to the next. This means that all modularity in the net has to be propagated to the nodes through the connections. Another way is to not simply create full connectivity between layers, but leave specific connections out. So by adding hidden layers without full connectivity, we can greatly enhance the amount of modularity in the network, without raising the number of weights to astronomical numbers. See for instance the network in figure 3 which can be separated in two parts.

Since there are no connections between the two parts of the network presented in figure 3, the number of weights is reduced by 10 compared to a fully connected network with the same number of nodes in each of the layers. Apart from a speed up of the learning time caused by less connections, there might also be a speed up due to the greater modularity of the network.

To further enhance this idea of modularity, we define a *module* to be a group of mutually unconnected nodes with as well the same set of input as of output nodes. This means that a fully connected network has the same number of modules as it has layers. The network from figure 3 has 6 modules. Every node in a backpropagation network belongs to exactly one of these modules. In figure 4 the network of figure 3 is remodelled to this standard and looks a lot simpler than its original counterpart.

To test whether this modularization worked, Boers and Kuipers [Boers92] implemented a backpropagation network for the XOR-problem (see also §7.1) with a different topology than the network used my Rumelhart and McClelland [Rumelhart86]. The two networks are shown in figure 5. Rumelhart and McClelland's network (figure 5a) got stuck in a local minimum a couple of times during their experiments. Boers and Kuiper found that their network (figure 5b) not only always learned to solve the problem, but it also learned faster than Rumel-
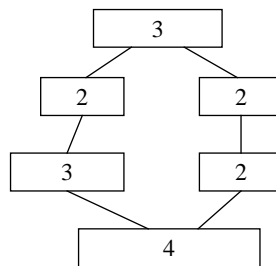

figure 4 A modular network

figure 5 Two networks that are capable of solving the XOR-problem

hart and McClelland's network did. On plotting the number of training steps required by the net as a function of the two learning parameters Boers and Kuiper found a much more regular dependence for their network than they found for Rumelhart and McClelland's network. Apart from that, they found that for the best combination for the learning parameters the Rumelhart and McClelland network needed 1650 training cycles, while the Boers and Kuiper network needed only 30 cycles. McClelland and Rumelhart also found a net with a different architecture, which did work with all experiments, although it still tended to be slow in learning.

The intuitive idea behind imposing modularity on a network is modelling the weight-space in such a way that all local minima disappear and making the error surface a lot smoother. This largely enhances learning.

A major problem modularity poses us with, is that of finding a suitable topology for a given problem (in determining whether a network is good, we also need to know what problem it is supposed to solve, since different problems have different suitable topologies). This was already a problem with classic networks, but with the extra complexity of the network connections, this problem gets even more difficult to solve, especially by hand. To solve this, one could for example try to modify the structure during training (see chapter 3) or one could (and that is more *biological plausible*, i.e. that resembles more the way that nature 'tackles' this problem) try to create a 'good' topology via genetically produced blue prints.

2.4.5   Activation functions

The activation function is the function that calculates the activation of an artificial neuron (unit). The most commonly used function is the *sigmoid* function. The (most) general form of a sigmoid function is:

$$f(stim_p) \ = \ \frac{act_{max} - act_{min}}{1 + e^{-stim_p}} + act_{min}$$

where $stim_p$ denotes the weighted stimulation the unit gets from its input units (the units that are connected **to** the unit) when the network is presented with input pattern p, act_{max} denotes the maximum and act_{min} denotes the minimum activation of an unit. Usually act_{max} is 1 and act_{min} is 0. This function is the function I used for the experiments (see chapter 7). Another widely used variant of this function uses *0.5* for act_{max} and *-0.5* for act_{min}.

Sometimes a *linear* activation is used. Then the activation of a unit for an input pattern p is defined by: $f(stim_p) = stim_p$. If this function is used, it is mostly only used for the output units.

The *gaussian* activation function keeps appearing in neural network applications and is defined by:

$$f(stim_p) = e^{-0.5\,(stim_p)^2}$$

A very special activation function is the *hyperbolic tangent* function defined by

$$f(stim_p) = \frac{e^{stim_p} - e^{-stim_p}}{e^{stim_p} + e^{-stim_p}}$$

The experiments presented in this research where obtained with the *sigmoid* function, but the backpropagation library that was written by Boers and Kuiper [Boers92] was modified in such a way that experiments can just as easy be done with any of these other functions or with a mixture of these functions (it is possible to specify the type the units in a module should have; all the units in a module are of the same type).

3   Algorithms that modify the structure

As stated in the previous chapter, one of the major problems with Feed Forward Neural Networks (FFNN) is the problem of finding a 'good' topology for a given problem. This is a difficult and delicate task. If the structure is too large, it will be able to learn the problem by heart and, as a consequence, hardly be able to generalize. If, on the other hand, it is too small, it won't learn the problem at all.

Over the past few years neural scientists have grown weary of their 'educated' guesses of the structure of the network and devised algorithms that try to find a 'good' topology. This chapter focuses on algorithms that change the structure of the network *during* training. In Chapter 5 a different approach that uses a *Genetic Algorithm* will be presented. In this chapter, after a brief discussion of two types of algorithms that modify the structure, two inspiring methods will be explained: The Cascade Correlation Learning Architecture, created by Fahlman and Lebiere [Fahlman88] and The Growing Cell Structures method, developed by Fritzke[Fritzke93].

3.1   Constructive versus destructive algorithms

Based on the hypothesis that the smallest net that is able to learn the training data will produce good results on data it is not trained with (i.e. will have a good generalization property), researchers came up with a number of algorithms. The algorithms that modify the structure during training, are usually classified in two types: *destructive* algorithms and *constructive* algorithms.

A *destructive* method begins with a net that is too large, and then reduces it. It removes nodes or weights from large, trained networks in order to improve their generalization performance ([Cun90], [Mozer89]). It continues to do so until,i.e. in most cases, the pruned net is no longer able to classify the training data correctly (usually the pruned net is shortly retrained). Then the previous net is taken to be the optimal, given the starting network. Destructive algorithms produce networks that generalize reasonable well [Omlin93], leaves us with the problem

that one should construct a starting network that is too large. Furthermore since the initial network will be large, a lot of training time will be necessary to train the initial network and retrain the intermediate (still too large) networks. Therefore these algorithms are relatively slow.

A special class of destructive methods is formed by methods that use a modified training scheme. The object is to find a net as simple as possible that learns the problem. So these methods try to minimize a function that depends on the error *and* on a complexity (called a penalty) term ([Hanson89], [Nguyen93]). This penalty term is usually a function over the weights so that the smallest weights will be forced to zero.

Constructive algorithms don't have the problem of determining the initial structure, they just start with the simplest structure possible. Besides the usual weight update rules, a constructive method should also define a criterion when (and where and how) to change the current topology of the net and how to assign initial values to the new weights. Typically, new resources (nodes, weights) are added to the structure so that it keeps previously acquired knowledge (c.q. weights). The process of training, adding and retraining stops when some criterion is met; e.g. the error is beneath an acceptable level.

Constructive and destructive methods for adapting the networks during training are complementary. Constructive methods can be used to find a network that responds well and destructive method can be used to improve the performance of an already trained network. So these methods could easily be glued together. First one can use a constructive algorithm to find a good topology, then one can use a destructive algorithm to optimize (if necessary) the topology. Two of the most interesting constructive algorithms are The Cascade-Correlation Learning Architecture [Fahlman90] and Growing Cell Structures [Fritzke91]. A brief explanation of these algortihms will be presented in the remainder of this chapter.

3.2   The Cascade-Correlation Learning Architecture

This constructive method for supervised learning was introduced by Fahlman and Lebiere in 1990 [Fahlman90]. It starts with only an input and an output layer which are fully connected. There is also a bias input, permanently set to +1.

If after a number of training cycles (with quickprop [Fahlman88]) no significant error reduction has occurred, then the network is tested on the test set to determine the error. If the error is below a predefined upper limit, the algorithm stops, if it isn't, there must be some residual error that should be reduced. So an hidden layer of one unit is added to the net. This new unit receives trainable input connections from the input layer and from all pre-existing hidden layers. These weights are trained in a special way (see below) and after that this layer will be fully connected to the output layer. Then these input weights are frozen and all the output weights are trained once again. This process is repeated until the error is acceptably small.

The training method for the new hidden layers input weights consists of a number of passes over the training set, adjusting the new unit's input weights

after each pass. The goal of this adjustment is to maximize *S,* the sum over all output units *o* of the magnitude of the correlation[1] between *V,* the new unit's activation value, and E_o, the residual output error observed at output unit *o*. *S* is defined as

$$S = \sum_o \left| \sum_p (V_p - \bar{V})(E_{p,o} - \bar{E_o}) \right|$$

where *o* is the number of the output unit at which the error is measured and *p* indicates the training pattern. The quantities $\bar{E_o}$ and $\bar{V}$ are the values of E_o and *V* averaged over all training patterns.

To maximize *S,* the partial derivative of *S* with respect to each of the new unit's incoming weights, w_i, is computed. So

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \bar{E_o}) f_p' I_{i,p}$$

σ_o is the sign of the correlation between the new units activation value and the error of output o, f_p' is the derivative for pattern p. Now a gradient ascent is performed to maximize S with quickprop (only the input weights of the new unit will be trained).
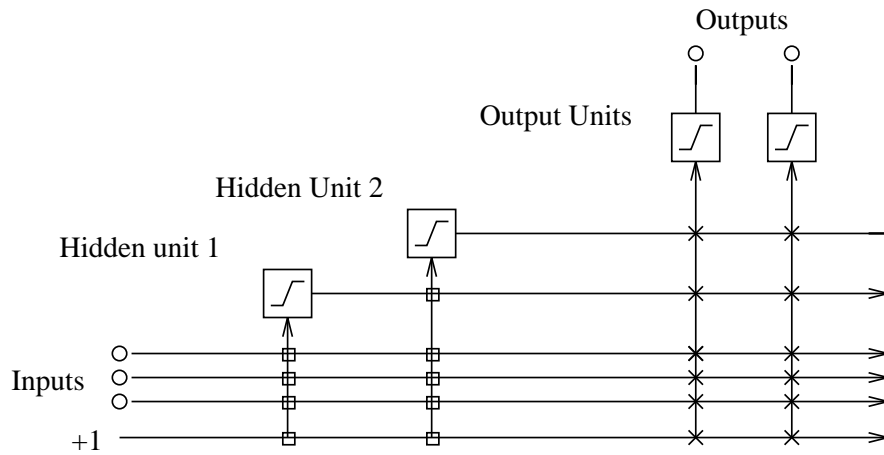
figure 6 The Cascade architecture after two hidden units have been added for a problem involving three input and two output units. The +1 denotes the bias. The vertical lines sum all incoming activation. Boxed connection are frozen (after installation), X connections are trained repeatedly [Fahlman91]

When *S* stops improving, the new layer (of one unit) is fully connected to the output layer and it's inputs weights are frozen for the rest of the training. Now the training of all the weights connected with the output layer starts again. Because the activation value of the hidden unit is correlated to the remaining error at the output units, that error can be reduced during the next training cycle

---

1. S is actually a covariance, not a true correlation (some of the necessary normalizations are omitted). In their early versions Fahlman and Lebiere used true correlation, but this version of S worked better in most cases.

since for input patterns that cause high errors in the outputs, the hidden unit will produce a high value.

To improve the usefulness of a new unit, a pool of candidate units are trained in parallel, each starting with a different set of random weights. The candidate unit that has the highest correlation score S is chosen as new unit to be added to the network.

One of the most remarkable features of this constructive method is that at any time, only one set of weights is trained (all incoming weights to the output layer or all incoming weights of the new hidden unit). So no error signals are propagated backwards through the network connections. This gives an opportunity to greatly speed up the algorithm. Since the incoming weights of already installed hidden unit never change, the activation values of this unit for all training patterns can be cached (if the target machine has enough memory). This can result in tremendous speed up, especially for large networks.

Although this method works rather well, it may lead to a network of great depth (many connected (small) hidden layers). To see this suppose there is a problem P that can be divided into two disjunct problems P_1 and P_2. If we use a Cascade-Correlation Learning Network (CCLN) to solve this, it will take at least two hidden layers, if both problems can not be linearly separated. Each hidden layer can only correlate well with one of the problems. A simple example of such a problem is the one presented in TABLE 1. The first output is the result of $input_1$ XOR $input_2$ and the second output of $input_2$ XOR $input_3$ (see also §7.1). This particular problem can be solved with only one hidden layer of two units. If we have a

TABLE 1.   A simple problem consisting of two disjunct problems

input1	input2	input3	output1	output2
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	0	1
1	1	1	0	0

Table 1. $Output_1$ is the XOR function of $input_1$ and $input_2$, output is the XOR funtion of $input_2$ and $input_3$.

problem P that consist of n such problems, then a CCLN will construct a network with at least n hidden layers, where one hidden layer with n hidden units would have been sufficient, if the inputs are directly connected to their corresponding outputs (see also §7.1.3). In both cases there are n hidden units but the CCLN has at least n-1 more connections, because it produces n hidden layers of one unit instead of one hidden layer of n units.

A small modification on Cascade-Correlation (CC) presented by Simon, Corporaal and Kerchkhoffs [Simon92] reduces the depth of resulting networks significantly. Instead of a hidden layer of only one unit, they add (if necessary) one hidden layer with as many units as there are in the output layer. Instead of trying to maximize the correlation between a unit and all the output units, their version tries to maximize the correlation between the unit and its corresponding output unit. Once the incoming weights for the new hidden units are trained and frozen, these units have to be connected to the output layer. They tried two methods: only connect the hidden unit to its corresponding output unit, or connect the hidden unit to all the output units. Both these methods produce networks with a smaller number of hidden layers and seem to generalize better then plain CC.

3.3 Growing Cell Structures.

Growing Cell Structures is a method inspired by Kohonen's features maps [Kohonen82]. This method is not a pure constructive method because it includes the occasional removal of units. But it starts with small cell structures and uses a controlled growth process. It is as an unsupervised learning algorithm [Fritzke91] but recently a modified version of the method was capable of supervised learning [Fritzke93]. Both these versions are worth to be looked into.

3.3.1 Growing Cell Structures and Unsupervised Learning

Before the network model is described, it seems appropriate to define exactly the kind of problems the network is supposed to solve. Suppose that every input is a real value. If the problem has n input signals then the input space V is equal to $V = \Re^n$. The input signals 'obey' an unknown probability distribution $P(\xi)$. The objective is to generate a mapping from V onto a discrete k-dimensional topological structure A. This mapping should have the following properties:

- It should be *topology-preserving*. That means that similar input vectors are mapped on topologically close (could be the same) cells of A and that topologically close elements in A should have similar signals being mapped onto them.

- It should be *distribution-preserving*. That means that every cell of A should have the same probability of being the target of the mapping for a random input vector according to the probability function P.

If the dimensionality of A is smaller than that of V and it is still possible to preserve the similarity relations, then the complexity of the data is reduced without loss of information.

The initial topology of the network A is a k-dimensional simplex. For $k = 1$ this initial structure is a line segment, for $k = 2$ a triangle, for $k = 3$ a tetrahedron. For $k > 3$ the simplex is called a hypertetrahedron. The *cells* (or units) are the $k + 1$ vertices of the simplex. The edges denote topological neighbourhood relations. Every cell c has an n-dimensional vector attached. This vector may be seen as the position of c in the input space.

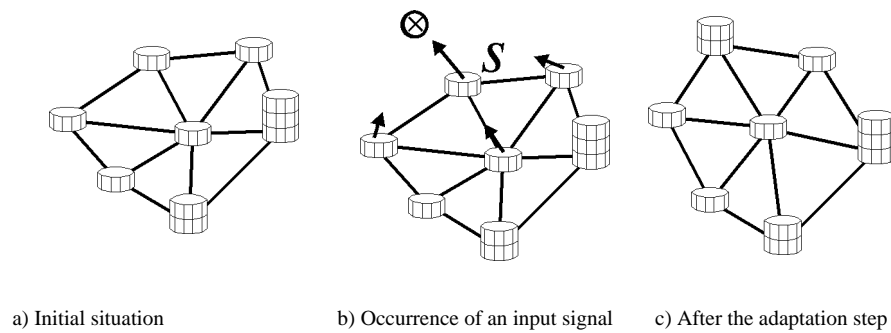| a) Initial situation | b) Occurrence of an input signal | c) After the adaptation step |

figure 7 One adaptation step for a two-dimensional cell structure. Only the best-matching unit and
its direct neighbors are adapted. The columns represent the signal counter values. The signal
counter of the best matching unit is incremented [Fritzke93].

When presented with an input vector the method determines the cell that is topo-
logically the closest to it (the euclidic distance is used). This cell, called the best-
matching unit, is moved a fraction in the direction of the input vector propor-
tional to the difference. The 'neighbours' of this best-matching unit are also
moved a, usually smaller, fraction. This is called an adaptation step.

Every cell has a region in the input space with the property that every element of
that region will be mapped onto that specific cell. This is known as the *Voronoi
region*. To simplify the computations it is assumed that the input space is an arbi-
trarily large but finite subregion of $\Re^n$. So every Voronoi region is finite. Even
with this simplification it is very hard to compute the size of a Voronoi region for
n greater than two. So if the size of a Voronoi region is to be computed (why is
described below), it is estimated by the size of an n-dimensional hypercube with
a side length equal to the mean distance between the cell and its neighbours.

The neighbours are being moved to keep (or to fulfil) the topology-preserving
property. How can the distribution-property be fulfilled? This property is ful-
filled if every cell has the same probability of being the best matching unit for a
randomly chosen input vector. Additional to moving the units around the algo-
rithm uses another (much more important) method of obtaining this property:
adding units to the structure.

To determine where to add (or to delete) a cell, every cell has a matching score
sometimes called a signal counter. In the simplest variant this indicates how
often a cell has been the best-matching unit. Since the cells are slightly moving
around, more recent signals should be weighted stronger than previous ones.
This is achieved by decreasing all (real valued) counter variables by a certain
percentage after each adaptation step. After a fixed number of adaptation steps
the cell with the highest matching score q is determined. Then the neighbour f of
q is determined whose associated input vector is the most distant to the associ-
ated input vector of q. Now a new cell r is created. It gets an associated input
vector that lies half way between the input vectors of q and f. This new cell is
connected in such a way that the structure remains a structure consisting only of
k-dimensional simplexes, i.e. it is connected to all the cells that are neighbours of
both q and f, q and f are no longer 'neighbours' (see figure 8). If this cell r would
have been present from the beginning of the process, some of the input vectors
that mapped onto one of its neighbours would have been mapped onto it. So the
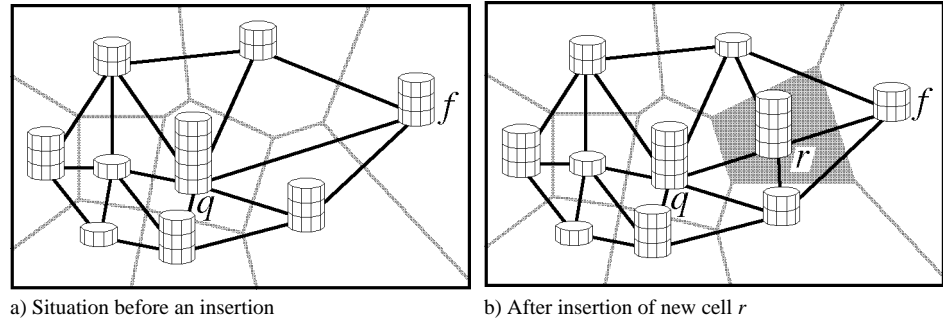
a) Situation before an insertion   b) After insertion of new cell *r*

figure 8 An insertion for a two dimensional cell structure. The cell *q* has received the most input signals so far (situation a. The columns represent the signal counter variables). A new cell *r* has been inserted and thus the Voronoi regions change (the grey lines denote the border of the Voronoi regions). The signal counter variables are redistributed according to the changes of the Voronoi regions [Fritzke93].

matching score for all its neighbours are lowered and the matching score for this new unit equals the sum of the losses of its neighbours. For each neighbour *c* the new matching score τ_c equals

$$\tau_c = \frac{\left\|F_c^{(new)}\right\|}{\left\|F_c^{(old)}\right\|}\tau_c \qquad \text{(EQ 1)}$$

where $\|F_c\|$ denotes the size of the *n*-dimensional volume of the Voronoi region F_c. Notice that insertion near a cell *c* decreases both the value of the signal counter and the size of its Voronoi field. The reduction of the Voronoi field makes it less probable that *c* will be best matching unit for future input signals.

Summarizing, the principal algorithm of Growing Cell Structures looks like this:

1. Start with a *k*-dimensional simplex, with the vertices at random positions in the input space
2. Perform a constant number of adaptation steps
3. Insert a new cell and distribute the counter variables.
4. If the desired net isn't reached go to 2.
5. Stop.

The probability function $P(\xi)$ could be the union of a few disjunct regions in the $\Re^n$. If that is the case then the algorithm will create cells with (almost) no chance of being the best matching unit (the cells that 'lie' between the disjunct regions). So a better structure could be produced if these cells were removed. To ensure that the structure remains a structure consisting of *k*-dimensional simplexes, all the simplexes this cell participated in are removed. This could lead to a removal of another cell.

An obvious problem for this method is the size of *k*. It should be equal to the number of components of the input vectors that are stochastically independent, but usually that is part of the problem. One doesn't know the complexity of the problem, one just wants it solved! Consider the extreme case that the input space is a subspace of $\Re^{100}$ but only two of the hundred components matter (i.e. are stochastically independent). Suppose that you gave *k* a value of 100 because you didn't know the internal dependencies. If now for one of the cells the mean edge
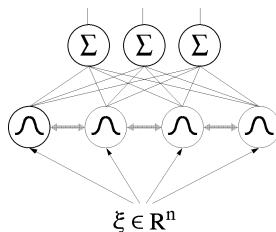
figure 9 A Supervised Growing Cell Structures network. The bold horizontal arrows between the Gaussian units mean that there are topological neighborhood relations among the Gaussians (in contrast to conventional RBF). They are used to interpolate the position of newly created Gaussians from existing ones as well as to define the radius of the Gaussian.

length shrinks by ten percent (because a cell is added to the structure), the size of its corresponding Voronoi region (and thus the counter) will be less than 0.03 promile of its previous size. Since it only depends on two components this does not reflect the change of the 'receptive field' of the cell very well. This may lead to the effect that most insertions occur in one region of the structure. This is due to the fact that a newly inserted cell gets nearly all the signals of its neighbours because the change of their Voronoi fields is overestimated.

3.3.2   Growing Cell Structures and Supervised Learning

The growing cell structure method can produce a lot of information of the input space, but what if one has a problem that consists of producing the right output vector given an input vector when only for a couple of input vectors the desired output vector is known? Such a problem is usually 'tackled' using a supervised learning technic (see also chapter 2). Assume for the rest of this section that the data of the problem consists of a number of pairs $\left(\xi_i \in \Re^n, \zeta_i \in \Re^m \right)$ where ξ_i is the input and ζ_i is the desired output of the i-th pair.

The modification of the growing cell structure method so that it can solve problems that involve supervised learning, has a lot in common with the Radial Basis Function network (RBF) [Moody88], but it eliminates some drawbacks of that approach.

Like RBF, the method presented by Fritzke [Fritzke93] uses a layer of units with Gaussian activation functions and an output layer of m outputs units with linear activation functions (figure 9). Each Gaussian unit c has an associated vector $w_c \in \Re^n$ indicating the position of the Gaussian unit in the input vector space and a standard deviation σ_c. The layer with Gaussian units is fully connected to the output layer.

The Gaussian units (see §2.4.5) correspond to the cells from the unsupervised method, so for a given unit c there is a set N_c that contains all its 'neighbours'. When presented with an input vector $\xi \in \Re^n$, the method computes the activation D_c of every unit according to

$$D_c(\xi) \ = \ e^{-(\|\xi - w_c\| / \sigma_c)^2}$$

The best matching unit (bmu, the unit that has the highest activation) will be moved a fraction in the input space in the direction of ξ, and it neighbours an even smaller fraction.

The activation of an output unit o_j is computed by $\sum w_{ci} D_c$ were D_c is the activation of the Gaussian unit c and w_{ci} denotes the weight of the connection between unit c and output i and the summation is taken over all Gaussian units c. The weights w_{ci} are updated according to the delta rule[1] since there is only one layer of weights.

The counting variable (the variable that is used to determine where to add a new unit) of the best matching unit is raised by the overall squared error between the actual output $o = o_1, ..., o_m$ and the desired output $\zeta = \zeta_1, ..., \zeta_m$

If the current problem is a classification problem, a different updating method is used. The counting variable of the bmu is raised by one only if ξ is classified *incorrectly*.

Networks built with this classification error as insertion criterion tend to be still very small when they start classifying all training examples correctly. This is due to the fact that new cells are only inserted in those regions of the input vector space where still misclassifications occur. On the other hand, learning does practically halt when no misclassifications occur anymore, even if the "raw" mean square error is still rather large. This can lead to poor generalization for unknown patterns. So it seems advisable to use a weighted combination of classification and mean square error.

Whenever a new cell r is inserted, it gets two vectors assigned to it: one position vector $p_r = (p_1, ... p_n)$ which denotes the place of the new unit in the input vector space and one output vector $w_r = (w_{r1}, ..., w_{rm})$. The output vector is not initialized with random values but it is obtained through a redistribution similar to that used for the counting variable of the new unit. The output vectors of its neighbours are lessened an amount equal to the estimated change in their Voronoi fields and the output vector of the new vector equals the sum of their losses (compare with the formula on page 21).

In doing this redistribution the new cell is given output weights such, that it will activate the output units in a way similar to its "mean" neighbour. Since the neighbouring Gaussians overlap considerably, the overall output behaviour of the network is not changed very much. In future adaptation steps, the new unit can develop different weights and contribute so to the error reduction in this area of the input space.

---

1. In such a case, quickprop can also be used. Because it works faster than the delta rule, Fahlman and Lebiere used Quickprop in a similar case (§3.2).

4   Evolution and Learning

Apart from trying to find algorithms that modify the structure of a neural network *during* training, researchers are also trying to construct a good neural network for a given problem with *Genetic Algorithms (GAs)*. Instead of producing a network structure directly with a GA, Boers and Kuiper used a system that produced a set of production rules based on L-systems. In this chapter L-systems and GA will be described briefly. The last part of this chapter consists of a description of the *Baldwin-effect*.

4.1   L-systems

L-systems were introduced in 1968 by Lindenmayer [Linden68] in an attempt to model the biological growth of plants. An L-system is a string rewriting mechanism and, in that sense, a kind of *grammar*. It is as opposed to traditional grammars a *parallel* string rewriting mechanism.

4.1.1   A simple L-system

A grammar consists of a starting string and a set of *production rules*. A production rule consists of a *left side, a 'transition symbol'* and a *right side*. The left side denotes the 'state' a (sub)string should be in, so the production rule can be *applied*, the right side denotes the state the (or part of the) substring is in after applying the production rule. The transition symbol is a separator between the two sides.

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. A production rule must first *match* before it can be applied; the left side of the rule has to be the same as the part of the string on which the rule is applied. Then a part of the string is replaced (or *rewritten*) by the right side of the rule. Whereas in other grammars
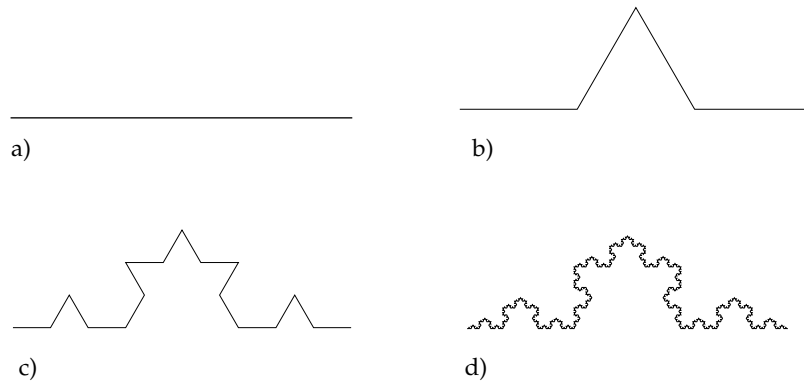
figure 10 The first steps of the Koch fractal. a) shows the axiom, b) the generator (production rule) c) after applying the production rule 2 times. d) after 5 times.

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

This parallel rewriting property of L-systems makes it possible to form a simple set of rules that are able to generate strings that, given the right interpretation are approximations of certain types of fractals. Take for example the Koch-graph [Koch05]. If one uses a LOGO-style turtle to interpret the strings generated [Szilard79] by the following L-systems, one would get intermediate stages of the Koch Graph (see figure 10):

$$\begin{aligned} \text{Axiom} &\quad F \\ \text{Production rule} &\quad F \rightarrow F - F + + F - F \end{aligned}$$

The traditional interpretation of the symbols used is: F, draw a line, and $+/-$, turn the direction of the turtle to the right/left (given a fixed angle).

4.1.2   Bracketed L-systems

A disadvantage of the turtle symbols from the previous paragraph, is that they can only make so called *single line drawings*, and since a lot of plants do have branches, it is not good enough to model the growing of plants. To give the turtle the freedom of movement to create branching, two new symbols are added: [remember the current position and direction of the turtle (push) and] restore the last stored position and direction (pop). With these two symbols, far more realistic drawings of plants can be made that follow strings produced by an L-system, as can be seen in figure 11a.

4.1.3   Context Sensitive L-systems

Another way of making more complex, more natural drawings of plants, is possible by introducing *context*. Context can be seen as a model of the exchange of information between neighbouring cells in a plant. It can be left, right or both for a certain string. An L-system with one sided context is denoted as a 1L-system and one with context on both sides as a 2L-system. A production rule is of the following form:
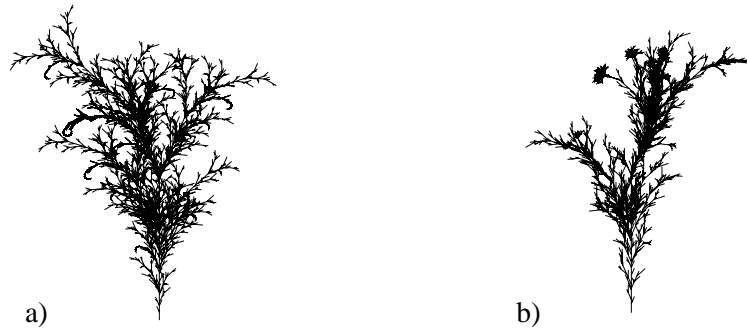
figure 11 Two plants generated by L-systems. a) This figure is obtained with a 'bracketed' L-system. b) This figure is obtained with a stochastic set of rules. For information on the set of rules that produced these artificial plants see [Boers92].

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

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

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

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

the string *ABC* would be rewritten to *XYZ*, after which neither of the rules applies. If we were to take these production rules:

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

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

Determining what the context, is a little more tricky with bracketed 2L-systems. Since the left and right context is not always direct left or right from the string or character that is to be replaced, but can be distanced by a bracketed pattern (these bracketed patterns would represent branches if we were to plot the string as a tree [Prunsik89]). If, for instance, we had a production rule with the following left side:

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

It could be applied on the *S* in:

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

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

4.1.4   Implementation

Prusinkiewicz and Hanan [Prunsik89] present a small L-system program for the Macintosh (in C). To experiment with L-systems Boers and Kuiper ported this to PCs [Boers92]. Besides fixing some "irregularities" the program was rewritten in order to accept less rigid input files. Two features were added: probabilistic production rules and production rule ranges (both from [Prunsik89]).

With probabilistic rules more than one production rule for the same *L*, *P* and *R* can be given, each with a certain probability. When rewriting a string, one of the rules is elected at random, proportional to its probability. This results in more natural looking plants, without them losing their characteristic appearance. The figure 11b shows a plant that was created with a set of probabilistic rules.

Production rule ranges introduce a temporal aspect to the L-system and tell which rules should be looked at during a certain rewriting step. This can be used for example, to generate twigs first and then the leaves and flowers at the end of those twigs.

4.2   Genetic Algorithms

4.2.1   Introduction

The Genetic Algorithm (GA) is a search strategy that operates on a population of chromosomes (also called individuals). Each chromosome contains an instance of the parameters of the problem to be solved, in some coded form. The goal of a GA is to generate a population such that the average of the fitness of the chromosomes of this population is an increasing functions, i.e. the population gets 'better' than the old population. To give the GA a meaning of 'better' or 'worse' every chromosome has a fitness value. A high fitness indicates a 'good' chromosome. The fitness value is calculated by a fitness function.

Two main operators of the GA are mutation and crossover. Mutation changes an arbitrary bit in the chromosome to introduce new instances of the problem. Crossover takes two chromosomes and exchanges some arbitrary parts, so properties of chromosomes are mixed. To generate a new population the GA selects chromosomes (the ones with higher fitness have a larger probability to be chosen), performs some operations on the chromosomes and copies these to the new population. To decide which chance a chromosome has in the selecting process, one can take the fitness value of the chromosome and divide it by the total fitness of the population. Another method is rank based selection: let the chromosomes be in descending (with respect to the fitness value) order. Suppose the first chro-

mosome (and since the chromosomes are sorted, it is the best chromosome) has a chance *c*. Now the second chromosome gets chance *c-s*, the third *c-2*s* and so on. (see [Whitley89]). For more information about GA see for example [Goldberg89].

4.3   Can Learning guide evolution?

Many organisms learn to usefully adapt themselves during their lifetime. These adaptations are often the result of an exploratory search, which tries out many possibilities in order to discover good solutions. It seems very wasteful of the evolutionary machinery not to make use of the exploration performed by the phenotype (the organisms) to facilitate the evolutionary search for good genotypes.

Some biologist have argued that nature does not waste these improvements but transfers information about the acquired characteristics back to the genotype. This is called the Lamarckian hypothesis. Nowadays most biologist don't think that evolution actually works that way, but that doesn't imply that learning can not guide evolution.

Suppose that the learned adaptations, improve the organisms chance of survival. Then the chance of producing offspring and hence of reproduction are also improved. The idea that learned behaviour could influence evolution was first proposed by Baldwin [Baldwin1896]. If specific learned behaviours become absolutely critical to the survival of individuals then there is selective advantage for genetically determined traits that either generally enhances learning, or which predisposes the learning of specific behaviours. At the very least, Baldwin's hypothesis indicates that learning will guide the direction of evolution. In its most extreme interpretation, the *Baldwin effect* suggests that selective pressure could make it possible for acquired, learned behaviour to become genetically predisposed or even genetically determined via Darwinian evolution.

Recently this effect got some new attention and not only from biologists. Hinton and Nolan showed how the performance of a genetic algorithm can be improved when it uses the Baldwin effect [Hinton87]. They used an extreme and simple example. Suppose that the problem is to find the minimum of a function that has a constant value for all vectors of the input space except for the goal vector. This is sometimes called a "needle-in-a-haystack" or "impuls function" problem. It is important to recognize the difficulty of this problem. Not only is there only one correct solution, but the result of every other input vector gives no information on where the correct answer may be.

An example of such a function is $F(a) = 1 - a_1 \times a_2 \times \ldots \times a_n$ where a is a n dimensional vector $(a_1, a_2, \ldots, a_n)$ and for every $i \in \{1, \ldots, n\}$, $a_i \in \{0, 1\}$. The GA used n genes[1], each controlling its own component of a. The values for

---

1. The actual problem was to construct a neural network of n connections, an individual was considered successful if and only if it has all connections correctly specified. This similar problem is used to simplify further reasoning
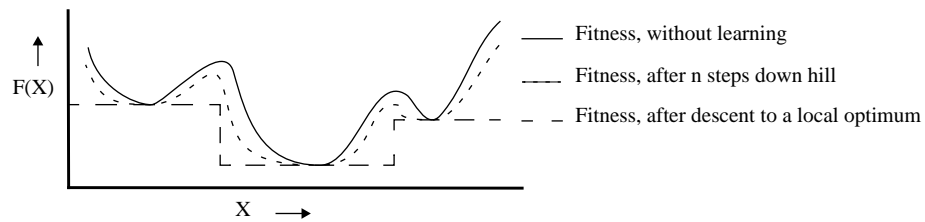
figure 12 How learning can modify the search space.

a gene could be 1 (if its corresponding component should be a 1), 0 (if its corresponding component should be a 0) and ? meaning that this component of *a* should be learned. The 'learning' method used was that each individual (phenotype) got a fixed number *G* of completely random guesses for the settings of their ? values. They were also given the ability to recognize that they have found the correct setting.

The fitness associated with each individual is higher when the actual number of guesses needed to find the correct setting is lower. So when an input vector is not the correct one, it still may give some information on where the correct answer may be. Because of this 'learning' strategy there is a basin of attraction around this 'needle' that the GA can use to move the population towards that needle.

An interesting point is that although theoretically eventually all ? should be replaced by 1's, this did not happen within 500 generations. Belew showed in analysis of this model [Belew89] that there is little selective pressure to replace ?'s in a string of almost all ?'s and that the same is true for replacing the last few ?'s. So the function of the number of ?'s in a population is a S-shaped curve.

In this extreme case not only the problem was quite hard but also the learning method used was not very sophisticated. Consider an unknown function F which one tries to minimize using a genetic algorithm [figure 12]. If each individual can move in the direction it 'thinks' the minimum lies, then the attractor basins are larger then if one did not use local optimization (e.g. learning).

Looking at figure 12 you can see both an advantage and a disadvantage of adding learning to a GA. The advantage is that 'obstacles' in the search space become easier to take and the attractor basins are larger. But as an additional effect learning may cause the search space near an optimum to flatten out, so that there is not much information on the exact location of the optimum. When the GA produces individuals that always learn the optimal adaptation then there is not so much need to find it by itself.

So learning can guide evolution but if it is to be effective in a computational environment, the speed up gained should be greater then the loss caused by the time the algorithm spends on learning. So to take advantage of this effect one must find a relatively simple learning scheme or a brilliant complex scheme that simplifies the search space dramatically. If you use for example a GA to find the best way to draw a graph, the learning scheme could contain a way to locally change the position of an edge.

Gruau and Whitley tried some different ways to help their genetic algorithm through learning [Gruau93]. They used a GA to find both the Boolean weights

and the architecture for neural networks that learn Boolean functions. Instead of coding the topology directly, they used a grammar tree to encode the network (*cellular encoding* [Gruau92]). The four 'modes of learning' they tested were:

- using a basic GA without learning

- using simple learning that makes some weights change sign (they used Boolean weights) only if the network is small

- using a more complex form of learning that changes weights on small networks, then "reuses" these weight in the development of larger nets[1] (developmental learning)

- using a form of Lamarckian evolution; learned behavior is coded back onto the chromosome of an individual

The results of their experiments showed that the GA converged much faster if it used learning then when it did not. Lamarckian and developmental learning where faster than the simple learning scheme. They suggested that, because they used a different training scheme, developmental learning is a bit like Baldwinian learning, so it can not be stated that Lamarckian learning is much faster then Baldwinian. They conclude that: "it is unclear, of course, whether our results pertaining to the *Baldwin effect* generalize to other domains, but this possibility is well worth exploration".

---

1. It is important to note that they used an unusual training scheme. One of the objectives was to find a grammar that could solve the parity problem [Gruau93] for the n-input case. Starting with n=2 the algorithm tries to find a solution, if it does the network building machinery is allowed an additional *recursive* step to parse the grammar and now the resulting network should try to solve the problem for n=3.

5 Local structure optimization

The genetic algorithm (GA) as developed by Boers and Kuiper [Boers92] tries to find a good neural network topology for a given problem. A produced network is trained a predefined number of cycles and then the total sum squared error of the net on the test set is used to determine the fitness (the smaller the error, the higher the fitness). If the GA produces a topology that results in a high error it only gets that type of information back. It does not get any information on *why* the proposed topology was such a bad one, or *where* in the structure, it has to add (or delete) some complexity (hidden units or weights). In this chapter some criteria will be developed to detect during training, where and if there are computational deficiencies in a modular neural network. Furthermore the different ways to add complexity are discussed and some methods to initialize the newly added complexity.

5.1 Criteria to detect computational deficiencies

Comparing *constructive* with *destructive* algorithms that modify the structure of a network (§ 3.1) you will notice, besides the differences in approach, a remarkable fact: where destructive algorithms put a lot of trouble in determining where to remove some complexity of the net (units and or weights), constructive algorithms just add complexity on a predefined place in a predefined way. Take for example Cascade Correlation (§3.2) [Fahlman90]: it will (not surprisingly) always produce a cascaded architecture, no matter what the complexity of the problem. Only the size of the 'cascade' is problem dependent. The algorithm as proposed by Marchand, Golea and Rujan only adds nodes to the hidden module of a network with one hidden module [Marchand90]. The Upstart Algorithm of Frean [Frean90] produces network structures that look like binary trees and the Tiling Algorithm of Mezard and Nadal [Mézard89] produces multi-layered networks where each new hidden layer has half the number of units of the previous one. Although it is not a fair comparison[1], Fritzke's method [Fritzke93] seems to

be the exception of this 'rule' (§3.3.2) i.e. that algorithm does not add nodes on a predefined place.

So where will we look to find criteria that detect computational deficiencies? First we will see why it is so interesting to have such criteria and then, after a discussion of the type of information that can be used for such criteria, a few possible criteria will be presented (§5.1.3).

5.1.1   Motivation

It is nice and easy to state that most constructive algorithms just add units in a predefined way on a predefined place, but why is it so interesting to know where to add complexity and is that possible? Well if a (modular) network can determine exactly where there are computational deficiencies and if it knew which kind of complexity (hidden unit or a connection) could diminish the problem then a modular constructive algorithm could be developed. Such an algorithm would combine the speed of a constructive algorithm with the advantages of a modular network.

To develop such a criterion is maybe one of the most challenging and difficult tasks concerning artificial neural networks, because it requires a high amount of knowledge of how these networks actually work and how they divide a problem in a few separate feature detectors.

But what if we had a criterion that was not so perfect that a constructive modular algorithm could be based on it, but still gave some indication on where to add complexity? Such a criterion could be used to exploit the *Baldwin effect* (§4.3) to guide a GA that produced modular networks.

5.1.2   Constraints on the criteria

There are a lot of possible ways to detect computational deficiencies. Even if we are going to settle for a less perfect criterion, it may be worthwhile to constrain 'the search space' of possible criteria. A few constraints such a criterion has to fulfill are:

- It should be reasonable simple to compute whether the criterion is met.

This is, of course, a very practical limitation and not a theoretical one. If the use of a criterion makes the process of adding complexity to the network too costly in relation to the changes it causes in the search space of possible topologies, it can not be used effectively to speed up the GA that tries to find good topologies.

- It should only use local information.

---

1. His approach is not directly comparable with 'normal' constructive methods for feedforward nets. For example, his Growing Cell Structure method for supervised learning does not use an input layer.
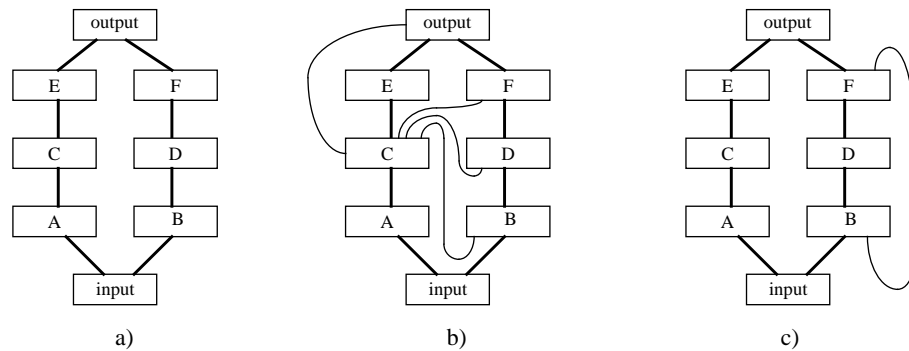
figure 13 Adding a connection between two modules. Figure *a* shows a basic modular feedforward network. If we connect module C (C the starting module) with module B, D, F or the output module this property is maintained (figure b). If however a new connetion is made between modules F and B in such a way that F is an input module for B the network is no longer a feedforward network (figure c).

This may seem very arbitrary and in a way it is, but it can be a great advantage if you are trying to use the criterion on a parallel computer. If you look at it in a modular sense then it becomes a bit clearer. A module is supposed to detect some (high level) feature(s) of the problem. It tries to compute this feature based on the information that the modules that are connected to it provide. If it can not compute the feature it has been trying to compute, it should be possible to determine that based on information that can be gathered by the module itself.

This brings along the somewhat related problem of what kind of complexity should be added. Until now, nothing was said about what kind of complexity should be added to solve the computational deficiency of a network. Ways of adding complexity to a modular network are:

- adding a module to the network

- adding a connection between two modules that are not connected yet

- adding a unit to an already existing module

Adding a module to the network is quite complex. One has to determine which modules should give information to this new module and which modules should get information from this module (and the size of the new module). In modular perspective, adding a module is adding a (higher order) feature detector, so the algorithm should decide if and how a new feature has to be computed. This is hardly a local or simple task.

To add a connection between two modules is not a local task. To see this look at figure 13. Imagine that the problem to be learned is much easier to learn if module C and F where connected. F is not a (distant) relative of C, so F cannot be considered as a module to connect module C to solely on information that is known to C (eq. information of the structure is necessary). If you want to use this method on a feedforward network you have to ensure that this does not introduce a cycle in the net.

Creating an additional unit in an already existing module, because the module has some computational deficiency, is a local task. Every module may decide for itself that it has such a deficiency. No global dependencies have to be checked; a feedforward network where one of its modules gets an additional unit, is still a feedforward network. This method of adding complexity to the network to solve computational deficiencies seems to fulfill the constraints better than the others.

The next paragraph will give some criteria to decide whether a module needs an additional unit.

5.1.3   Criteria to add a unit to a module

So the existing constructive methods cannot help much in finding criteria to detect locations of computational deficiencies in existing modular networks (because they add nodes or hidden layers on a predefined place, see the first alinea of §5.1). What about destructive algorithms? Some of these methods try to force small weights to zero (for example [Nguyen93]). If a hidden unit does not receive any input signals or does not give any output signals it is removed completely. Omlin and Giles [Omlin93] presented a pruning algorithm that prunes the hidden units with the smallest input vector. Well if small incoming weights indicate that a unit of a module is not that important, maybe large incoming weights to a module do indicate that the module can not cope with all the work, that means: the module is too small. This gives as a possible criterion:

- To decide whether a module is too small one can look at the size of the module's input vector, if that is large, the module is assumed to be too small and a unit is added to it. I will refer to this as the *in-weight*-criterion.

This is, of course, a very simple criterion. An algorithm based on this criterion is bound to be of little complexity and since a module uses the weights of its incoming connection to determine its activation vector, it is locally computeable. But it is a rough criterion, it does not use any information on the error produced by that module. A network with a module that has a large input vector, may be perfect for one problem and a disaster for another problem.

This leads to a more general type of criterion. If a network has not (yet) learned a specific problem it produces errors in the output units. These errors are then propagated back through the net. Back-propagation tries to reduce the errors caused by a module *M1* in module *M2*, by changing the weights between *M1* and *M2* (§2.4.2). If the output weights of a module change all the time, then it apparently it does not what it should do and it may be an indication that the module is too small. Thus:

- To decide whether a module is too small one can look at the amount of weight change on the outgoing connections of the module, the *out-moment*-criterion[1].

_____

1. It is called the moment criterion because the moment calculated in back-propagation with momentum equals these weights changes.

A module tries to detect some feature of the problem and when it has a lot of error, one could argue that it did not (yet) succeed to detect some useful feature of the problem. It may well be that the feature or features it tries to learn require more hidden units to produce a useful classification. So as a criterion also the amount of weight change of the incoming weights may be used:

- To decide whether a module is too small one can look at the amount of weight change on the incoming connection of the module, the *in-moment*-criterion.

The *in-* and *out-moment* criteria, use the absolute amount of weight change, independently of the current size of the weight. One could argue that a connection whose weights change all the time between *0.5* and *2.5* indicate more trouble than a connection whose weights change between *11* and *13*. With this in mind it is easy to think of two criteria

- To decide whether a module is too small one can look at the relative amount of weight change on either the incoming or the outgoing connections of the module. This will be called the *in-relative* and the *out-relative* criterion, respectively.

There are a lot of differences between 'real' neurons and their artificial counterparts. One of the more prominent ones is the fact that in nature a synapse (connection) is either inhibitory or excitatory and although the strength of the inhibition or the excitation may vary, it will never change from an inhibitory to an excitatory synapse or vice versa. When back-propagation is used as learning method, it is possible that a connection between two units changes its sign. If this happens then the 'relation' between those units is changed. If, during training, the signs of the connections from module M_1 to M_2 change a lot, then it seems reasonable to assume that either M_2 needs more hidden units to classify 'its' feature(s) or M_1 needs more hidden units to compute the difficult function M_2 requires. Generalizing to more modules this leads to two criteria

- To decide whether a module M is too small one can look at the number of times the sign of the connections between M and its output modules (the *out-sign*-criterion) or its input modules (the *in-sign*-criterion) changes.

In early simulations I noticed that the *sign* criterion and the *moment* criterion differed quite a lot on which modules needed another unit. So I tried a more messy approach and combined both these criteria into one criterion

- To decide whether a module M is too small, one can look at both the number of sign changes as to the absolute amount of weight changes of the connection between M and its output modules (the *out-combi*-criterion) or its input modules (the *in-combi*-criterion)

5.2   Installing a new unit

If we have decided that a unit should be added to a module *X* (figure 14*)*, we still have to decide how we are going to do that. This includes deciding how the new
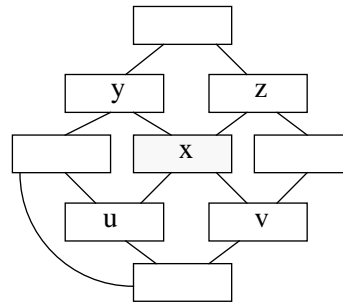
figure 14 An example of a modular network. Suppose we want to add a new unit to module *x*. We then have to make additional connections between all the units of the module's input modules (*u* and *v*) and the new unit and between the new unit and all the units of the modules output modules (*y* and *z*). Furthermore we have to initialize these new weights.

weights are determined, how and if the old weights of the module are changed and how and if the old weights of the network are changed. It may seem unnecessary to change the old weights. That it can be advantageous to change the old weights can be seen by the following argument. Suppose a module tries to classify a certain feature but fails one unit to be able to do so. Since the feature is not correctly determined, the whole network has modified its weights to compensate for this 'fault'. Now if we add a new unit to the 'troubled' module, it can correctly classify the feature and the network should try to 'forget' the adjustments it made to compensate for the 'fault'. It may well be that the network with the too small module converged to a local minimum during training. If this is the case, adding a unit to that module will not be enough to ensure that the module will learn the feature. The net first has to 'jump' out of the local minimum. While this may be too difficult for the learning algorithm, a temporary 'shock' may do the trick. To help the net to 'jump' out of the local minimum two methods were tried: *modshock* and *netshock*. With the modshock-method small random numbers are added to the incomming weights of the module that just has been changed (the module that lacked computational power and has obtained an additional unit). The netshock method does the same but now for the entire network: small random numbers are added to all the incomming weights of all the modules in the network. In the implementation of the software two parameters specify the upper limits on the absolute amount a weight may change when using those shock-methods. If that number *l* is specified then all the weights w_{ij} of module *j* (so the incomming weights of a module are modified) are changed according to:

$$w_{ij}' = w_{ij} + R\left(\frac{l}{\sqrt{units_j}}\right)$$

where w_{ij}' denotes the new value of w_{ij}, $units_j$ is the number of units in module *j* and *R(x)* is a function that returns a random value between *-x* and *x*. Depending on the setting of those parameters only the weights of the module that has been enriched or those of the whole network are changed this way.

Althoug the few tests that I did indicate that the size of these 'shock' parameters should be quite large (around 2 for a network hidden layers of 10 units) to produce a real increase in the error, the best overal results were obtained when I kept them small (around 0.1), see also chapter 7.

The simplest method to prevent the network of staying stuck in a local minimum is, of course, to replace all the weights with new random values. In a standard constructive algorithm this method is not a good one. All the previously acquired information is lost and the problem should be learned again. When used in combination with a genetic algorithm (GA) this method is one of the safest (and the slowest): if a network structure, that was acquired during the GA process, solves the problem well, you can be reasonably sure that this structure, when it is slightly modified, will give 'good' results when trained. This will be referred to as the *INIT*-method (initialize).

Another obvious method is to keep the old weights and only randomly set the new weights. This method will be referred to as the *KEEP*-method. This method is more desirable when used in a direct constructive algorithm (as opposed to an algorithm involving a GA). The information acquired during the previous learning phase will not be wasted, although it maybe necessary to 'shock' the network a little so that it may jump out of a local minimum.

Instead of setting the weights of the new unit randomly, one could try to figure out good initial weights for these connections. One could use a method like the one used with Cascade Correlation [Fahlman90] (§3.2). This method tries, by changing the weights of the input connections of this new unit, to maximize the correlation between the activation of the unit and the error in its output modules, and installs the outgoing weights according to that correlation. This will be referred to as the *CasCor*-method. One could also use a method like the one Fritzke uses to initialize the weights from the *RBF*-units to the output units [Fritzke93] (§3.3.2). This method takes for the new weights of the (new) unit the mean of its neighboring weights (the weights between the in- and output modules of this unit and its neighbors), the *Mean-Neighbour* (or *MN*) -method.

5.3   Network Dynamics

A special data structure is used to be able to use the different criteria. Besides the current weight and its movement (moment §2.4.2), the number of sign changes and the absolute amount of weight change of a connection are being stored. 'Absolute amount of weight change' means that solely the size of the weight changes are summed and not the signs of these changes. A weight that changes -*0.3* gets the same addition to its datastructure as a weight that changes +*0.3*.

Suppose all the connections of a certain module *X* went through a lot of change some time ago, but settled recently into a stable state. Then it may be possible that the module *X* meets the criterion to add a unit to *X* even though *X* is in a stable state. Such a module should not get an additional unit. In order to prevent that a large amount of weight changes during the beginning of the training period influences too strongly the decision if a module should get an additional unit, the absolute amounts of weight- and sign-changes are multiplied by a constant m with $0 < m < 1$. The computations that are performed during a training step for a certain weight w_{ij} are (see also §2.4.2):

$$\Delta w_{ij}(t+1) \;=\; \alpha \delta_{o,j} h_i + \beta \Delta w_{ij}(t)$$
$$w_{ij}(t+1) \;=\; w_{ij}(t) + \Delta w_{ij}(t+1)$$
$$abs(t+1) \;=\; \left| \Delta w_{ij}(t+1) \right| + m\ abs(t)$$
$$sgn(t+1) \;=\; m\ sgn(t) + (sign(w_{ij}(t+1)) = -sign(w_{ij}(t)))$$

where *abs(t)* is the absolute amount of weight change on time *t* and *sgn(t)* is the structure in which the 'number' of sign changes are kept. The *sign(x)* function returns a 1 if $x > 0$, –1 if $x < 0$ and *0* if $x = 0$. The second part of the computation for $sgn(t+1)$ is a *boolean* expression, a expression that can either be 1 or 0 (True or False).

Now we have developed criteria to see if a module needs an additional unit, how can we use them? To determine if a certain module needs an additional unit an *instability score* is computed. This score is just the sum of all the incomming or outgoing (depending on the criterion) connections, computed following the method implied by the criterion, normalized for the number of incoming or outgoing connections. Now it would be nice to have a formula *k* that depending on the used criterion and the topological properties of the module *M* (size of the module, number of connections etc.) could return a value *c* in such a way that *M* lacks computational power if and only if the instability score $s_M > c$. I tried several formulas but they did not seem to work as desired. So the algorithm was modified a bit so that if the network has not yet, after a number of training steps, learned the problem, the module with the highest instability score is determined and that module gets an additional unit. The instability scores are normalized for the number of connections to prevent large modules from getting all the additional units.

6   Implementation

During the research two different existing programs were changed and adapted. The first one (`backmain`) can be used to test the different criteria and the different ways to install a new unit. The second one (`genalg_w`) can be used to find 'good' network structures to solve a given problem.

This chapter describes the software. Hopefully it will offer some help if someone wants either to use the program itself or some ideas behind it.

6.1   Enviroment

The software is written to work on Unix based machines. Large parts of the software are written in C, but some, especially the backpropagation parts, are written in C++. The compiler was GNU C++. The simulations were run at the department of Computer Science of Leiden University on several SUN machines; models ELC, LPC, IPX and CLA. The final version of the software can easily be ported to other computers which have both C and C++ available.

6.2   The data files

People, who want to device an algorithm that solves a problem, have to decide how they are going to present that problem to their algorithm. Many scientists who work with neural networks, have all deviced their own strategies to present the problem to their algorithms. This made it difficult to use the same data for more algorithms and so to compare their results. Both the programs `backmain` and `genalg_w` use the *CMU Neural Network Learning Benchmark Data File Format*. A second datafile is used by `backmain` to specify the initial structure of the network.

6.2.1   The CMU Neural Network Learning Benchmark Data File Format

The reason I dedicated a subsection to this data file format is twofold: it is the format I used and by dedicating a piece of this thesis to it, I hope to advocate its use. If more people use it, it will be a lot easier to compare results and to test a new problem with different algorithms. It is developed at the Carnegie Mellon University of Pittsburgh, USA. Besides the description file (see below) of the data format and C code to 'parse' the data files, a bench mark collection is accessible via anonymous FTP[1]. To describe the data format, the following explanation is provided (available at that FTP-site):

CMU Neural Network Learning Benchmark Database Data File Format

Maintainer: neural-bench@cs.cmu.edu

This file contains a description of the standard format for data files in the CMU learning benchmark collection. These files are a supplement to the benchmark description files that comprise the CMU Benchmark Collection. Data files are associated to their appropriate description file with a '.data' extension to the file name of the benchmark description.

SEGMENTS

Each data set is composed of two to four segments. The first segment is always the $SETUP segment. The $SETUP segment is immediately followed by the $TRAIN segment. There are also optional $VALIDATION and $TEST segments.

$SETUP

The $SETUP segment describes the requirements of the network to the program. Included in this segment is information on how the program should read the actual data segments as well as what type of inputs and outputs are required. All lines in the $SETUP segment should end in a ';'.

PROTOCOL: {IO, SEQUENCE};

The protocol parameter tells the program how to read the data sets included in the file. In an IO mapping, each vector of input and output values is a seperate training case, independant of all others. The network's output depends only on the current inputs. In a SEQUENCE mapping, the input/output vectors are presented in sequential order. The output may depend on earlier inputs as well as the current input vector.

OFFSET: <n>;

This appears only in SEQUENCE mappings. It is the number of input vectors to read before an output should be produced. For most problems, this will be set to '0'.

INPUTS: <n>;

This is the number of items in an input vector. However, since some data types, such as enumerations, may require more than one input unit to represent, the actual number of input units may be greater.

OUTPUTS: <n>;

---

1. The neural-bench Benchmark collection. Accessible via anonymous FTP on ftp.cs.cmu.edu [128.2.206.173] in directory /afs/cs/project/connect/bench. The email contact is: "neural-bench@cs.cmu.edu", use email if you want to donate data or you have encountered some problems. The data sets in this repository include the 'nettalk' data, 'two spirals', protein structure prediction, vowel recognition, sonar signal classification, and a few others.

Similar to INPUTS, this specifies outputs instead of inputs. Again, due to some data types requiring more than one unit to represent, there may be a disparity between this number and the actual number of output nodes in the network.

IN [n]: < CONT {Domain}, BINARY, ENUM {list} >;

Each input must have an explicit entry describing it. This entry contains the node number (n) and a data type. Available types are: CONTinuous, BINARY, and ENUMerated. Continuous inputs are floating point numbers with a specified domain (where all numbers are guaranteed to fall). Binary inputs have either a value of '+' or '-'. An enumerated input is one of a list specified within '{}'.

OUT [n]: < CONT {CoDomain}, BINARY, ENUM {list} >;

Each output must also have an explicit entry describing it. This entry contains the node number (n) and a data type. A CONT output is a floating point output which is guaranteed to fall within a specified CoDomain. BINARY outputs should have either a positive, '+', or a negative, '-', value. An ENUMerated output is one of the specified list.

NOTE - While listing node types, no fields are acceptable. In other words, the definition 'IN [1..13]: BINARY' or 'OUT [1]: ENUM {A..Z}' would NOT be legal.

$TRAIN, $VALIDATION, $TEST.

These segments contain the actual training, validation and testing data to be used by the network. The validation and testing segments are optional. Entries into one of these segments should have the form: <input 1>, <input 2>, <etc> => <output 1>, <output 2>, <etc>;

In SEQUENCE data sets, there may also be '<>' delimiters. These specify the end of a sub-sequence. Data sets on opposite sides of one of these delimiters should not affect eachother. The sequence-end delimiters do NOT require a semicolon, as do not segment headers. In data sets with an offset, there should be no arrow and no outputs for the first (n) inputs. Simply end the list of inputs with a semicolon.

COMMENTS -------- While no actual comment card is specified, any text occuring on a line after a semicolon should NOT be parsed. Therefore it is possible to comment a data file by starting a line with a semicolon.

The use of this format is straight forward. In figure 15, a small example is given of a problem in the CMU Data Set Format. The problem specified by it, was used to test each addcriterion (§ 7.1.2). In this research only problems with the IO protocol were used. So no problems were the ouput of the current input may depend on the previous input, were used.

6.2.2   The matrix file format

The program `backmain` needs at least two parameters. Besides the name of the datafile that contains the problem to be solved it needs the filename of a file containing an adjacency matrix of the network. The file should start with the number of units in the (initial) network and then for every unit a line with zeros or ones.

A zero on the *i*-th place on a row in the adjacency matrix means that there is no connection between the unit and the *i*-th unit, a one means there is. The program itself recognizes if two units are in the same module (if the i-th row is the same

```
;XOR3,XOR2 Data Set              $TRAIN

$SETUP
                                 +, - , + => - , +;
PROTOCOL: IO;                    -, - , - => - , -;
OFFSET: 0;                       -, - , + => + , -;
INPUTS: 3;                       -, + , - => + , +;
OUTPUTS:2;                       -, + , + => - , +;

                                 +, - , - => + , +;
IN [1] : BINARY;                 +, + , - => - , -;
IN [2] : BINARY;                 +, + , + => - , -;
IN [3] : BINARY;

OUT [1]: BINARY;                 !E!O!F!
OUT [2]: BINARY;
```

figure 15 A simple example of the CMU data format. The first output is positive (+) when only
one of the inputs is positive. The second output is the XOR function of the first two inputs. (The
file starts in the left column and continues in the right column)

as the *j*-th row, *i* not equal to *j* and the *i*-th colum is the same as the *j*-th colum
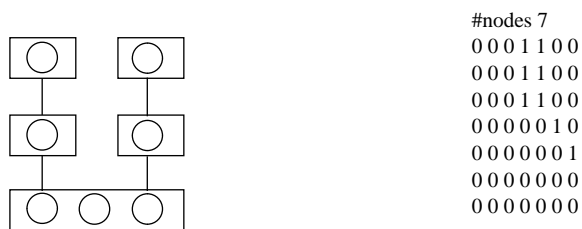then the units *i* and *j* are in the same module).

The second program (genalg_w) does not need an initial network structure,
since it creates network structures. It uses this structure internally to calculate the
fitness of a proposed network.

6.3   Parameters

6.3.1   Parameters of the test program (`backmain`)

Besides the two (needed) command line parameters (the filenames for the file
with the initial network structure and a CMU data file) it is possible to use a few
others with backmain. The main ones are:

- -C*mode*, where mode is the criterion to use to add a node: moment,
  sign, combi, relative or weight (§5.1.3). Additional you can
  specify the program to look at the input (in), the output (out) or both
  type of connections (inout). So if -Cinoutweight is used as


```
#nodes 7
0 0 0 1 1 0 0
0 0 0 1 1 0 0
0 0 0 1 1 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

a) A modular network.   b) The adjacency matrix of the network of fig. a

figure 16 An example of the relation between a network structure and the (adjacency) matrix file.
Figure a is the graphical representation of the network specifiec by figure b. Input units have no
incoming connections and therefore the first three columns are filled with zeros (three inputs).
Output connections have no out going weights and thus the last two rows are 'empty'.

parameter for the program, then the sizes of a modules in- and out-
putvector is used as criterion to add a unit to that module.

- `-M`mode, where mode is the way to install the new unit in the network:
  `init`, `keep`, `cascor` or `mean` (§5.2).

- `-i`# with # the number of times the network is trained before a module
  is located to add a unit to.

- `-m`# with # the shocksize of the changed module. That is the highest
  amout of weight change the weights connected to this module get, when
  an new unit is added to this module, If the install mode `init` is used,
  then this parameter is ignored.

- `-n`# with # the additional shocksize for all the modules.

- `-r`# with # the optional seed for the program. If it is omited a random
  one is used. This makes it possible to reproduce testresults.

- -t# with # the maximum number of nodes to add.

6.3.2   Parameters of the main program (`genalg_w`)

The program first reads 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 key-
word. An example simulation file, containing all the valid keywords is shown in
figure 17.

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 simula-
tion 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 #pres-
sure influence the genetic operators. #steps (maximum number of rewriting
steps) and #axiom are used by the L-system. #datafile is the data file in the CMU
benchmark format (the problem) to be solved. If #matfile is present then every
time a 'better' structure is found, the topology of that network will be saved in
matrix file format in the file specified. If #netfile is presented (in the example
above it is viewed as comment) not only the structure but the weights of all the
connections as well will be saved in the specified file. #nrofiter tells the program
how many training cycles it has to perform on every (usefull) network structure.
If #trainingsec is present, then the #nrofiter parameter is neglected. It indicates
how may seconds the program may train a (usefull) network structure. When this
parameter is used, smaller network structures are favoured above larger ones,
because they can do more training cycles in an equal amount of time then the
larger ones. #addcriterion, is the criterion on which the decision is based to add a
unit to an already existing module (see §5.1.3). #insertmethod is the method
used update the network when the a module gets an additional unit (see §5.2).

Multiple computers can run the same simulation simultaneously. Each program
opens the main population file and creates a small local population file contain-
ing a specified number of newly created strings. Then each program processes

```
##simulation file for suns
##

#population /home/mborst/SUN/bpcmu/gen/spiral/population
#control    /home/mborst/SUN/bpcmu/gen/spiral/control
#resultfile /home/mborst/SUN/bpcmu/gen/spiral/results
#path       /home/mborst/SUN/bpcmu/gen/spiral/

#size       100

#pmut       0.01
#pcross     0.5
#sites      2
#pinv       0.7
#pressure   1.4

#axiom      A
#steps      6

#datafile   /home/mborst/SUN/testsamples/two-spirals.data
#matfile    /home/mborst/SUN/bpcmu/gen/spiral/best.mat
##netfile   /home/mborst/SUN/bpcmu/gen/spiral/spiral.net
#nrofiter   500
#times      5
##trainingsec 2
#modshock   0.1
#netshock   0.1
#insertmethod cascor
#addcriterion inrelative
```

figure 17 Example of a simulation file. This one was used to find a good topology for the two-spiral problem. For the meaning of the various parameters see the surrounding text

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.

6.4   Back-propagation

Boers and Kuiper wrote their implementation of the back-propagtion algorithm in C++ [Boers92], because "an object oriented approach lent itself admirably to the implementation of modular networks". For this research their implementation is enriched with several features. The most important modifications deal with the criteria to add and the installation of additional units[1] to a module. Another one is the possibility to define for each module the activation function to be used. In this research only a standard sigmoid function was used, where the activation of a unit can vary between 0 and 1.

The implementation of back-propagation uses three classes: network, module and connection. The class network uses the others to implement the algorithm.

---

1. In this resarch only criteria where developed to decide which module lacked computational power, but not how much. So, although the implememtation makes it possible to add more units, if a module lacks computational power, only one unit is added.
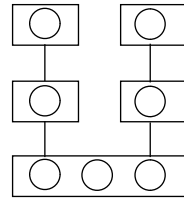
```
unsigned mod[]={3,1,1,1,1,0};
// module sizes; 0 marks end.
conSpec con[]={{0,1},{0,2},{1,3},{2,4},{0,0}}
// connections; {0,0} marks end.
class network net(3,2,mod,con);
// First parameters are input and output size
```

a) A modular network;   b) C++ code to create the network of a)

figure 18 Creating a network. The simple network of a) is created by the C++ code of b). The first array specifies the sizes of the modules. The first module receives number 0, the second 1, etc; a zero marks the end. The second array specifies the connections between the modules; {0,0} marks the end of the 'list' of connections.

There are three ways to create a network. The simplest way is to make two arrays, one specifiying the size of each module, the other specifying the connections between the modules (figure 18). Another creation method uses the size of and a pointer to an adjacency matrix (§6.2.2), to construct a network. The last one creates a network, from a file containing an adjacency matrix (§6.2.2) or a previously saved network.

7   Experiments

This chapter presents some results acquired by the developed software. Most of these are tests. These tests were used to see if the different ways to decide if a module needs an additional unit make some sense. Some results of the test on xor related problems will be shown and further the TC problem, 'where' and 'what' categorization, mapping of $[0.0,1.0]^2$ values onto four categories and the two spiral problem.

7.1   Some tests with XOR related problems

To be able to decide if the units are being assigned to the right modules, some knowledge of what these 'right' modules are is (of course) necessary. In 1969 it was proven that a network able to solve the XOR-problem should have a hidden layer [Minsky69]. The standard XOR function (eXclusive OR) is a boolean function of two variables, see TABLE 2.

TABLE 2.		The XOR function

input$_1$	input$_2$	output
0	0	0
0	1	1
1	0	1
1	1	0

If there are no direct connections between the input nodes and the output node then the hidden layer should have (at least) two nodes, otherwise the network will not be able to learn this problem. To see this, it is sufficient to see that if the output node only receives information from one hidden unit then the problem should already be solved at the hidden unit and then the XOR-problem would be solvable without hidden units and since it is not, a network that wants to solve the XOR-problem without connections between the input nodes and the output
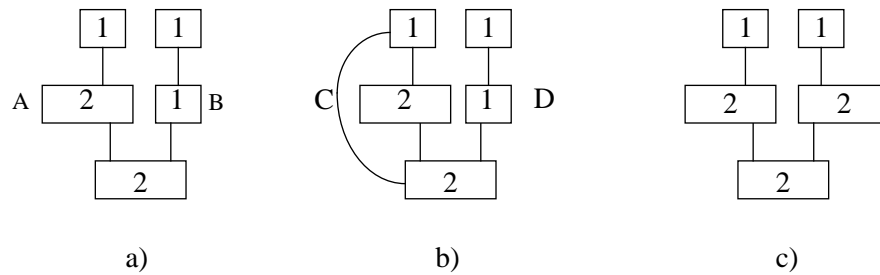
figure 19 Starting networks for the 2XOR test. Figures a and b show the initial structure of the starting nets used to test if the algorithm adds a node to the right modules. c) is the desired structure for a. This structure is capable of solving the 2XOR problem.

nodes, should have a hidden layer consisting of at least two units. Using this knowledge I developed a few simple related tests for the adding of nodes to already existing modules.

7.1.1   The 2XOR test

This is a very simple extension of the XOR-problem. Instead of one there are two output units. The two output units should both learn the XOR-problem. Two starting networks where tried on this problem.

This problem was merely used as a fast indicator if the developed method worked or not. If for example the program starts with the initial network of figure 19a, it should add a node to module B, because the subnetwork of module A has the computational power to solve the XOR-problem. It is important to note, however, that although the subnetwork of module A has the computational power to solve the XOR-problem, it does not always succeed (see for example [Rumelhart86]). The subnetwork C of figure 19b always learned the XOR-problem. All the methods passed the test of figure 19b (i.e. the methods indicated that a node should be added to module D). In the next two tables (TABLE 3 and TABLE 4) some results obtained with the network of figure 19a (the meaning of the data is explained below) are displayed.

The data of these tables were obtained by running the stand alone program `backmain` (see chapter 6). The program started with the network of figure 19a. The methods used to determine which module needs an additional unit (in this case module B) should be able to do so, in not that many training steps, because if we want to use such a method in combination with the genetic algorithm, extensive training of each net would result in a slow convergence. That, in combination with the fact that the differences between the different installing methods are greater when the number of training cycels is small, are the reason why the network was trained 250 times before the program made its decision to which module it should add a node. After the addition the trainingset was presented another 250 times. Each experiment was done 3 times, each with its own *randseed* and the tables show the average values. The *randseeds* were used to be able to compare the results. With the randseeds the random generator was initialized so that all the selection methods started with the same initial weights. As with all the other experiments the network was trained to produce an output of 0.9 when it should be true and with 0.1 if it should be false.

The *Selection method* is the method used to determine which module should get an additional unit (see §5.1.3). The *Installing method* is the method used to install the newly created module in the network (see §5.2). The *size of A* and the *size of B* denote the (average) size of modules A and B of figure 19a. So a size of 2.33 means that in two of the experiments the module had two units and 3 units in only one experiment. The *SQE* means the Sum sQuared Error (see §2.4.2). The *#cor. round* means the number of correctly learned test cases (and training cases with these experiments), *round* means that output values above 0.5 were treated as 0.9 and output values lower than 0.5 were treated as 0.1. The first table (TABLE 3) shows the results when the selection methods are used with the *IN*coming weights of a module and the second table (TABLE 4) shows the results when the selection methods use the *OUT*going weights of a module to determine if its computational power is insufficient.

TABLE 3.   Results of the various methods on the 2XOR problems (IN)

Selection method	Installing method	size of A	size of B	SQE	#cor. round
COMBI	CASCOR	2.00	2.00	0.31	4.00
COMBI	INIT	2.00	2.00	1.12	4.00
COMBI	KEEP	2.00	2.00	0.34	4.00
COMBI	MEAN	2.00	2.00	0.33	4.00
MOMENT	CASCOR	2.00	2.00	0.31	4.00
MOMENT	INIT	2.00	2.00	1.12	4.00
MOMENT	KEEP	2.00	2.00	0.34	4.00
MOMENT	MEAN	2.00	2.00	0.33	4.00
RELATIVE	CASCOR	2.00	2.00	0.31	4.00
RELATIVE	INIT	2.00	2.00	1.12	4.00
RELATIVE	KEEP	2.00	2.00	0.34	4.00
RELATIVE	MEAN	2.00	2.00	0.33	4.00
SIGN	CASCOR	2.33	1.67	0.66	3.67
SIGN	INIT	2.33	1.67	1.40	3.67
SIGN	KEEP	2.33	1.67	0.68	3.67
SIGN	MEAN	2.33	1.67	0.66	3.67
WEIGHT	CASCOR	2.33	1.67	0.66	3.67
WEIGHT	INIT	2.33	1.67	1.20	3.67
WEIGHT	KEEP	2.33	1.67	0.66	3.67
WEIGHT	MEAN	2.33	1.67	0.68	3.67

Except for the *in-sign* and the *in-weight* selection method, the *in* selection methods seem to work fine. The *CasCor* method seems to be the best way to install a new unit (it produces the Least Error) for this problem. Next is the *Mean*-method, closely followed by the *Keep*-method. It may come as no surprise that

(with 250 training cycles) the *Init*-method produced the biggest error (the information learned during the training of the imperfect net was lost).

TABLE 4.   Results of the various methods on the 2XOR problem (OUT)

Selection method	Installing method	size of A	size of B	SQE	#cor. round.
COMBI	CASCOR	2.33	1.67	0.4	3.67
COMBI	INIT	2.33	1.67	0.98	3.67
COMBI	KEEP	2.33	1.67	0.42	3.67
COMBI	MEAN	2.33	1.67	0.41	3.67
MOMENT	CASCOR	2.33	1.67	0.4	3.67
MOMENT	INIT	2.33	1.67	0.98	3.67
MOMENT	KEEP	2.33	1.67	0.42	3.67
MOMENT	MEAN	2.33	1.67	0.41	3.67
RELATIVE	CASCOR	2.33	1.67	0.4	3.67
RELATIVE	INIT	2.33	1.67	0.98	3.67
RELATIVE	KEEP	2.33	1.67	0.42	3.67
RELATIVE	MEAN	2.33	1.67	0.41	3.67
SIGN	CASCOR	2	2	0.31	4
SIGN	INIT	2	2	1.12	4
SIGN	KEEP	2	2	0.34	4
SIGN	MEAN	2	2	0.33	4
WEIGHT	CASCOR	3	1	1.09	3
WEIGHT	INIT	3	1	1.33	3
WEIGHT	KEEP	3	1	1.09	3
WEIGHT	MEAN	3	1	1.09	3

With the *out* selection methods (see TABLE 4), the results are quite different. All the selection methods except the *out-sign* and the *out-weight* methods produced only in two out of three trials the expected network structure. The *out-weight* method did not even produce the expected network once. The *out-sign* method produced in all three trials the desired structure. For the installing methods the same can be concluded as before; the *CasCor* method is best, followed by the *Mean*-method, the *Keep*-method and the *Init*-method, in that order.

7.1.2   The XOR3 XOR2 test

This XOR-based problem is comparable to the 2XOR problem. Now one of the outputs has to learn the XOR function of three inputs and the other just the XOR function of two of them. I mean by the XOR function of three inputs, the function presented in TABLE 5. The function results only in true if just one of the inputs is true, otherwise the function returns false. This test is slightly more complex than the previous one and I tried it because I expected that the 3XOR function as described above, would require a hidden layer of three units if there were no direct connections between the input module and the output module, but the results of the experiments (see below) proved me (terribly) wrong and thereby

TABLE 5.   The XOR function of three inputs.

input$_1$	input$_2$	input$_3$	output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

showed once again that to construct a 'good' network structure by hand is a complicated task.

For these experiments I used the same approach as with the 2XOR problem: three experiments for every *in* and *out* selection method in combination with every installing method, the training set was presented 250 times, before an additional unit was added and 250 times after the last one was added. As starting network I used an input module (3 units), two 'hidden' modules (both of 1 unit) and two output modules (see figure 20a). Each 'hidden' module is connected only to its output module and there are no connections between the input module and the output modules.

The results of these experiments are comparable with the ones obtained with the 2XOR test from the previous paragraph. If the selection method finds a 'good' structure, then the *Cascor* method results in the lowest error. All the selection methods, except the *in-weight* and the *out-weight* methods, found an appropriate network structure to solve this problem (see figure 20).

7.1.3   The 4XOR test

In Chapter 3 I claimed that the standard Cascade Correlation Learning Architecture strategy made too complex structures if the problem to be solved consists of numerous independent problems. To verify this statement I tried the following problem. There are 5 inputs and 4 outputs and the first output has to compute the
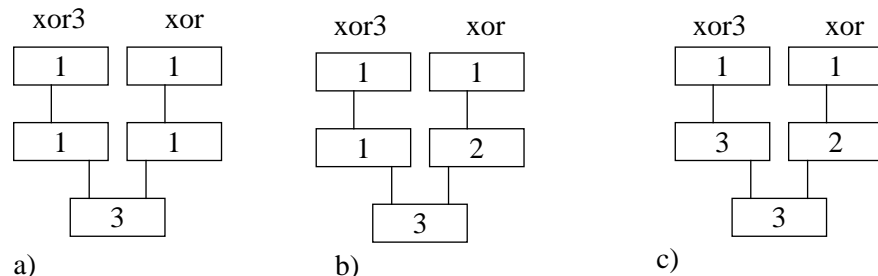

figure 20 Networks for the XOR3XOR2 problem, a) the starting network. b) the network that all the selection methods except the *in-weight* and the *out-weight* methods produced. This network is able to classify the test set correctly if rounding is used, but still has a large SQE. The smallest error was obtained with the network of figure c).
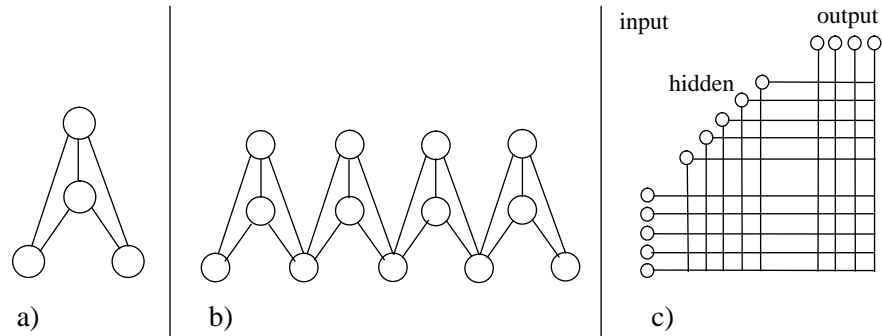
figure 21 Networks for the 4XOR problem. Since the network of figure a can solve the standard XOR problem, the network of figure b can solve the 4XOR problem. Figure c is the smallest network Cascade Correlation (CC) came up with. Note that a crossing of two lines indicate that the corresponding units are connected (the bias unit is not drawn). The solution of CC has 75 connections whereas the solution of figure b has only 20 and a hidden unit less.

XOR function of the first two inputs, the second output has to compute the XOR function of the second and the third output and so on. I tried to 'solve' this problem twenty times with the Cascade Correlation program[1] and although it should be possible to solve this with only 4 hidden units, Cascade Correlation produces 16 times a structure with 5 hidden units and four times with 6 hidden units.

7.2   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 [Rumelhart86]). 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). The eight possible 3x3 patterns and a sample of such a pattern on a 4x4 grid are shown in figure 22. The sample pattern is one of the 32 input patterns for 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.

The method of Boers and Kuiper found the network of figure 23a. They compared it with standard back-propagation networks (networks with one hidden
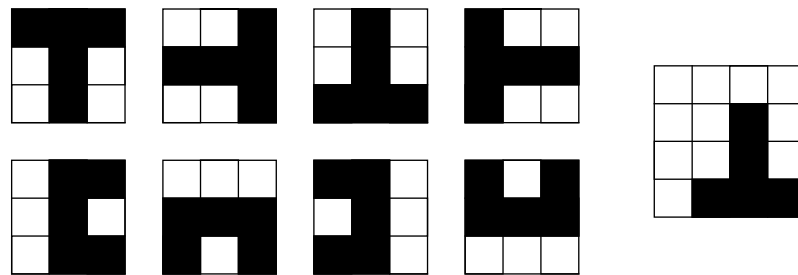

figure 22 The 8 letter orientations and one sample of an orientation in a 4x4 grid.

---

1. The C code was re-engineered by Matt White of the code of Scott Crowder, who based his code on the Lisp version of Scott Fahlman.
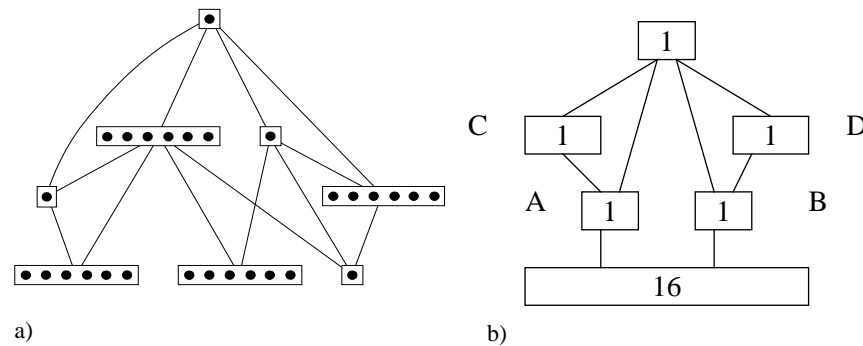
figure 23 Network structures for the TC problem: a) was produced by the algorithm of Boers and Kuiper. Note that the structure only uses 13 of the 16 inputs. b) the starting network used with this experiment.

layer). Their network outperformed the standard networks with a reasonable margin (out of 50 trials their network did classify after 250 training cycles the test set in 45 cases out of 50 where the best of the standard nets (one hidden layer with 6 units) did classify the test set only in 32 cases.

In this research the network of figure 23b was used as a starting network. If *Keep*, *Mean* or *CasCor* were used as installing method than after adding one or two units, the network classified the patterns correctly, independently of the selection method! If the network was trained *500* times instead of *250* times, the same can be said over the *Init* method: all the selection methods produced structures that learned the network after one or two units had been added. The starting structure of figure 23b is almost a good enough structure. When I allowed the network to keep adding modules even after the previous structure had solved the problem, a large variety of structures were produced. Most of the times first one or two units were added to one of the modules *C* or *D* and after that three or four units were added to the modules *A* or *B* (see figure 23b). Two of the most successful structures (structures that produced the least error) are (numbers indicate the number of hidden units in module *A*, *B*, *C* and *D* respectively) 1-5-1-3, 4-1-1-5 and 3-3-3-1).

7.3   'Where' and 'What' categorization

Another problem that was tried by Boers and Kuiper [Boers92] was proposed by Rueckl et al. [Rueckl89] 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 (figure 24a), 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. [Livingst88]). 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 24). Of importance was the number
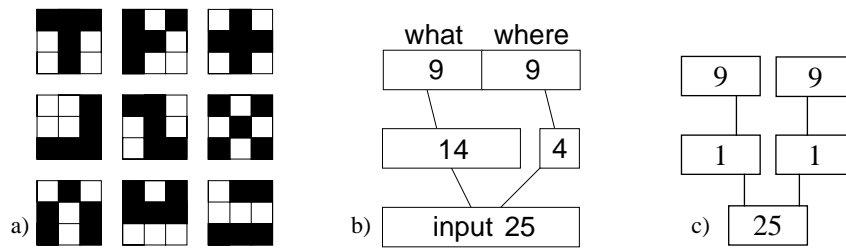
figure 24 The 'what' and 'where' problem. a) the 9 patterns. b) the optimal network according to Rueckl et al [Rueckl89]. c) the starting network used for `backmain` (see chapter 6).

of nodes allocated to the form and place system respectively. The optimal network found is shown in figure 24b, 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).

In this research this problem was tried to see if the proposed selection methods (criteria) were able to find the 'optimal' distribution of units between the two hidden units if it was forced to use them. To do so I used the starting network of figure figure 24c and tried all the selection methods combined with the *Init* installation method[1]. Every selection method had to solve this problem 4 times. For every experiment the same 4 random initializing values or *randseeds* were used, to be able to compare the results.

When the network was trained *500* times all the selection methods except the *in-sign* and the *out-sign* methods, produced network structures that learned the problem within *500* training cycles. The networks produced by the *in-weight* and the *out-weight* criterion tended to be somewhat larger and performed a bit worse than the one produced by the *in-* and *out-, combi*, *moment* and *relative* criteria.

Although in some trials these criteria came up with the structure found by Rueckl et al. (see figure 24b), they produced more often a network that has a hidden module of 13 units for the 'what' part and 6 units for the 'where' part. Remarkable is that all the methods that performed well started with adding 3 units to the hidden module for the 'where' part. Then the 'path' to the ideal structure differs for the different trials and methods, but in most cases the 'where' part still got a fourth additional unit before the 'what' part got its first additional unit. A possible explanation for this is that when both the hidden module for the 'what' part and the 'where' part still have one unit, a higher error reduction can be obtained by adding a unit to hidden module of the 'where' part. It is interesting to see the path the different methods follow. To accomplish that, every network between 1 unit for both the 'where' and the 'what' module and 18 for both the 'where' and the 'what' module were trained 10 times with 300 training cycles and the average SQE was plotted (see figure 25). Another remarkable

---

1. The *Init* method was used to ensure that the performance of the network was a cause of the structure and not caused by the 'extra' training the other weights got after adding an additional unit.
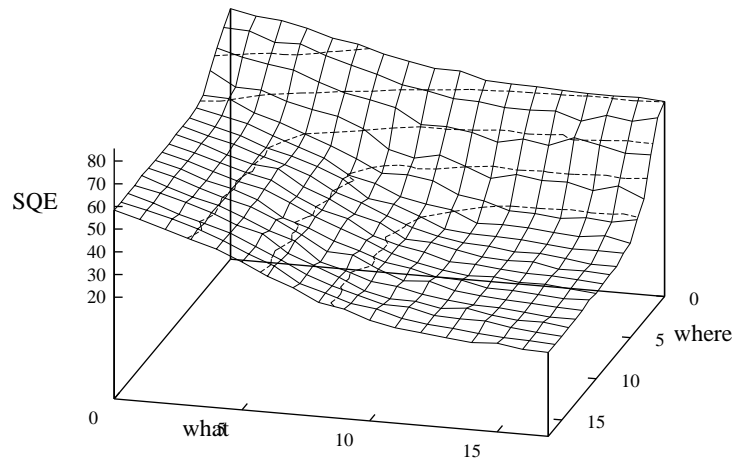
figure 25 The average error per structure. The 'where' axis denotes the number of nodes in the hidden layer of the associated subnetwork. The 'what' axis shows that number for its subnetwork. The SQE axis denotes the average error after 300 training cycles over 10 trails.

fact was that when I repeated this experiment but trained the network *5000* times instead of *500* times the results where almost exactly the same, although the SQE was generally a bit (but not much) smaller.

7.4   Mapping problem

This problem is more difficult for standard back-propagation 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, Boers and Kuiper used 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 26a. 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 disturbed by that noise (and therefore ignored them), Boers and Kuiper 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. Boers and Kuiper tried this experiment hoping their method would find a small, modular network that was able to learn the second mapping correctly. It took six days and 11 Sun Sparc4 workstations to converge to the network shown in figure 26b. It was found after roughly 85,000 string evaluations.
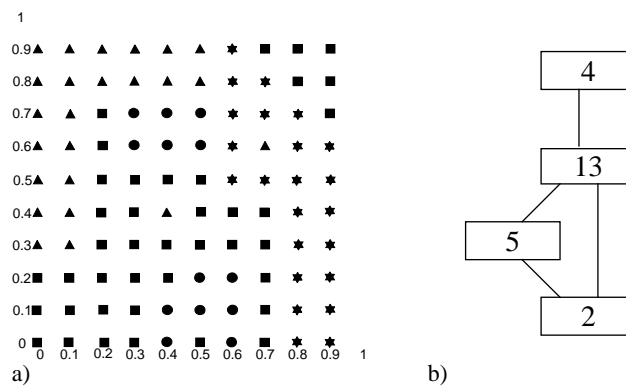
figure 26 The mapping problem. a) the second mapping as introduced by Van Hoogstraten. b) the 'optimal' structure found by the algorithm of Boers and Kuiper [Boers92].

In comparison to the 2-20-20-20-4 network used by Van Hoogstraten, the network from figure 26b 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 26b had an error of 14, versus 74 for the 2-20-20-20-4 network.

When I tried this problem I used (again) the ideal network structure of Boers and Kuiper and brought the number of units in a hidden module back to one. The same conclusion as from the other experiments could be drawn with the exception of the *in-sign* and the *out-sign* selection methods. These methods were in this experiment the best, the *in-sign* method came three times out of 4 trials up with a 2-2-13-4 (the input is connected to both hidden layers, the connections between the modules is the same as with the network of figure 26b) network that classified all the patterns correctly. The error was on average 13.5. Surprisingly enough in a lot of trials there were networks with substantial lower errors (around 4) but these networks qualified 'only' 98 or 99 out of the 100 patterns correctly.

8 Conclusions and recommendations

8.1 Conclusions

One of the great advantages of neural networks embodies implicit also a great disadvantage: since you do not have to teach these systems exactly how to solve a problem, it is very hard to tell how these networks come to their solutions. Not only for the scientific community but also for the people who are confronted with the solutions provided by neural networks, it would be interesting to understand a little more about how these neural networks function.

One of the most difficult tasks (for a human) is to determine if and how the structure of an existing neural network should be changed, in order to improve its performance. The 'logical' way to tackle this problem is to use a computer to solve this problem. The scientific community tries very hard to think of useful strategies. Some use methods that modify the structure of a neural network during training (see chapter 3, for example [Fahlman90] [Frean90]), others use the example of evolution in nature to produce 'good' artificial neural network structures to solve a given problem (see chapter 4, for example [Boers92] [Gruau93]). In general the methods for finding 'good' structures during training produce problem independent structure types and the ones for finding 'good' structures by an evolutionary approach take too much time to run effectively on large problems.

The method presented by Boers and Kuiper [Boers92] uses a *recipe*. Instead of coding all the units and their connection, they coded just a set of rules that produces a *modular* network stucture.Their results showed that modular network structures learn quicker and better then non modular networks for a couple of problems, but it still takes their algorithm a lot of time to come up with 'good' structures for larger problems.

Adding a local optimization method can speed up a genetic algorithm (see for example [Hinton87]). The objective of this research was to find a local optimization method that could be used with the algorithm of Boers and Kuiper. The opti-
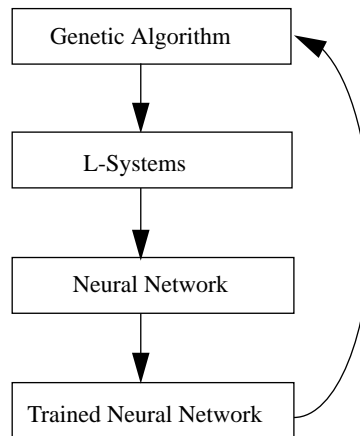
figure 27 An overview of the method of creating a trained neural network with the algorithm of Boers and Kuiper.

mization method that resulted from this research adds a unit to an already existing module to help that module, and thus the network structure as a whole, to overcome computational deficiencies. A few criteria to see if a module lacked computational power were developed (see §5.1) and tested with a couple of ways to treat the weights of the existing network after adding a unit to a module (see §5.2). The experiments showed that the *CasCor* method was the best way to install a new unit in the network. The criteria where applied on the **incoming** weights and on the **out**going weights. Those criteria based on the incoming weights produced in most cases far better results then the same criteria based on the outgoing weights. Of those criteria the *in-relative*, *in-moment* and the *in-combi* criteria where the best to use to decide which module should get an additional unit. The *in-sign* criterion was too instable (in some experiments it produced unacceptable results, in other experiments it produced very good results) and the *in-weight* criterion is not usable, only in a few trials it produced acceptable results.

8.2 Further research

During this research only one method of locally optimizing a modular neural network structure was tried. Although there were good reasons to do this (local optimization should be fast and local), it may still be interesting to try other approaches. One could for example try to find a criterion that indicates if two unconnected modules should be connected. Besides adding computational power, one could try to remove computational power as well, although that can be 'dangerous' in a network that has not yet reached a (local) minimum. Dangerous because maybe with the connection it could solve the problem. When combining the local optimization method with the algorithm of Boers and Kuiper it takes too long to let all the structures train until they reach a (local) minimum.

Further one could try, of course, other criteria to decide if a module lacks computational power. For example one could look at the total amount of error within a module to decide if it has a computational deficiency or not. In other parts of the algorithm (see figure 27) one could also do a lot of research. Especially the GA.

References

[Baldwin1896] J.M. Baldwin; 'A new factor in evolution'. In *American Naturalist*, 30:441–451, 1896.

[Belew89] R.K. Belew; 'When both individuals and populations search: Adding simple learning to the Genetic Algorithm'. In *3th International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.

[Bloom88] F. Bloom and A. Lzerzon; *Brain, Mind and Behavior*. Freeman, 1988.

[Boers92] E.J.W. Boers and H. Kuiper; *Biological metaphors and the design of modular artificial neural networks*. Master's thesis, Leiden University, 1992.

[Cun90] Y. le Cun, J.S. Denker and S.A. Solla; 'Optimal brain damage'. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, 598–605, Morgan Kaufmann Publishers, 1990.

[Darwin1859] C. Darwin; *The Origin of Species*. John Murray, 1859.

[Dawkins86] R. Dawkins; *The Blind Watchmaker*. Longman, 1986. Reprinted with appendix by Penguin, London 1991.

[Fahlman88] S.E. Fahlman; 'Faster-Learning Variations on Back-Propagation: An Empirical Study'. In *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, 1988.

[Fahlman90] S.E. Fahlman and C. Lebiere; 'The Cascaded-Correlation Learning Architecture'. In *Advances in Neural Information Processing Systems II*, 524–532, Morgan Kaufman Publishers, 1990.

[Fahlman91] S.E. Fahlman and C. Lebiere; *The Cascaded-Correlation Learning Architecture*, CMU-CS-90-100, School of Computer Science Carnegie, Mellon University, Pittsburgh, PA 15213, 1991.

[Frean90] M. Frean; 'The Upstart Algorithm: A method for Constructing and Training Feedforward Neural Networks'. In *Neural Computations*, *2*, 198–209, 1990.

References

[Freeman91]   J.A. Freeman and D.M. Skapura; *Neural networks: algorithms, applications and programming techniques.* Addison-Wesley, Reading, 1991.

[Fritzke91]   B. Fritzke; 'Let it grow — Self-Organizing feature maps with problem dependent cell structures'. In *Proc. of the ICANN-91 Helsinki*, 1991.

[Fritzke93]   B. Fritzke; *Growing Cell Structures — A Self-organizing Network for Unsupervised and Supervised Learning.* TR-93-026, 1993.

[Gazzaniga89]   M.S. Gazzaniga; 'Organization of the human brain'. In *Science*, *245*, 947-952, 1989.

[Goldberg89]   D.E. Goldberg; *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, 1989.

[Gruau92]   F. Gruau; '*Cellular encoding of genetic neural network*'. TR 92.21, Laboiratoire de l'Informatique pour le Parallélisme, Ecole Normale Supérieure de Lyon, 1992.

[Gruau93]   F. Gruau and D. Whitley; *Adding learning to the cellular developmental of neural networks: Evolution and the Baldwin effect.* In *Evolutionary Programming*, 1993.

[Hanson89]   S.J. Hanson and L.Y. Pratt; 'Comparing Biases for Minimal Network Construction with Back-propagation'. In *Advances in Neural Information Processing Systems 1 (NIPS 88)*, 177–185, San Mateao, California, Morgan Kaufmann, 1989.

[Happel92]   B.L.M Happel; *Architecture and function of neural networks: designing modular architectures.* Internal Report, Leiden University, 1992.

[Heemsk91]   J.N.H. Heemskerk and J.M.J. Murre; 'Neurocomputers: parallelle machines voor neurale netwerken'. In *Informatie*, *33-6*, 365-464, 1991.

[Hinton87]   G.E. Hinton and S.J. Nowlan; 'How learning can guide evolution'. In *Complex Systems*, 1: 495–502, 1987.

[Hoehfeld91]   M. Hoehfeld and S.E. Fahlman; *Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm*, CMU-CS-90-130, School of Computer Science Carnegie, Mellon University, Pittsburgh, PA 15213, 1991.

[Hoog91]   R.J.W. van Hoogstraten; '*A neural network for gentic facie recognition*'. MSc Thesis, Leiden, 1991.

[Koch05]   H. von Koch; 'Une methode geometrique elementaire pour l'etude de certaines questions de la theorie des courbes planes'. In *Acta mathematica*, *30*, 1905.

[Kohonen82]   T. Kohonen; 'Self-Organized Formation of Topologically Correct Feature Maps'. In *Biological Cybernetics*, 43, 59–66, 1982.

[Linden68]   A. Lindenmayer; 'Mathimatical models for cellular interaction in development, parts I and II'. In *Journal of theoretical biology, 18*, 280–315, 1968.

[Livingst88]   M. Livingstone and D. Hubel; 'Segregation of form, color, movement and depth: anatomy, physiology and perception. In *Science*, *240*, 740–749, 1988.

[Marchand90]   M. Marchand, M. Golea and P. Ruján; 'A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons'. *Europhysics Letters*, *11*, 487–492, 1990.

[Mézard89]   M. Mézard and J-P Nadal; 'Learning in Feedforward Layered Networks: The Tiling Algorithm'. In *Journal of Physics A*, 22, 2191–2204, 1989.

References

[Minsky69]	M. Minsky and S. Papert; *Perceptrons*. MIT Press, Cambridge, MA, 1969.
[Moody88]	J. Moody and C. Darken; 'Learning with Localized Receptive Fields'. In *Proceedings of the 1988 Connectionist Models Summer School*, 133–143, 1988.
[Mozer89]	M. Mozer and P. Smolensky; 'Skeletonization: A technique for trimming the fat from a network via relevance assessment'. In *Advances in Neural Information Processing Systems*, 107–115, Morgan Kaufmann Publishers, 1989.
[Nguyen93]	H. Nguyen; *Automatic Determination of Neural Network Architecture*, MSc Thesis, Univeristy of Leiden, 1993.
[Omlin93]	C.W. Omlin and C.L. Giles; *Pruning recurrent Neural Networks for Improved Generalization Performance*. Revised Technical Report No. 93-6, April 1993, Computer Science Department, Rensselaer Polytechnic Institute, Troy, N.Y., 1993.
[Parker85]	D.B. Parker; *Learning logic*. MIT Press, Cambridge, MA, 1985.
[Prunsik89]	P. Prunsikiewicz and J. Hanan; '*Lindenmayer Systems, Fractals and Plants*'. Springer-Verlag, New York, 1989.
[Prunsik90]	P. Prunsikiewicz and A. Lindenmayer; '*The algorithmic beauty of plants*'. Springer-Verlag, New York 1990.
[Rueckl89]	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.
[Rumelhart86]	D.E. Rumelhart and J.L. McClelland (Eds.); *Parallel distributed processing. Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.
[Simon92]	N. Simon, H. Corporaal and E. Kerckhoffs; *Variations on the Cascade-Correlation Learning Architecture for Fast Convergence in Robot Control*, Delft University of Technology, Electrical engineering Department, 1992.
[Szilard79]	A.L, Szilard and R.E. Quinton; 'An interpretation for D0L-systems by computer graphics'. In *The Science Terrapin*, *4*, 8–13, 1979.
[Werbos74]	P.J. Werbos; '*Beyond regression: new tools for prediction and analysis in the behavioral sciences*'. Unpublished Ph.D. thesis, Harvard University, Cambridge, MA, 1974.
[Whitley89]	D. Whitley; 'The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best'. In *Proceeding of the 3rd International Conference on Genetic Algorithms and their applications (ICGA)*, 116–121, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.