Diplomarbeit

A Genetic Algorithm for the Topological Optimization of Neural Networks

Rafał Sałustowicz
Matr. Nr.: 127242

Technische Universität Berlin
Fachbereich 13 – Informatik
Institut für Wissensbasierte Systeme (WBS)

Aufgabensteller:	Prof. Dr. E. Konrad
Betreuer:	Prof. Dr. D.E. Rumelhart

Abgabe: 18. April 1995

Abstract

This study focuses on the genetic construction of feedforward neural networks. Feedforward neural networks constitute a good solution for many difficult problems. They can be trained with fast and reliable learn algorithms once their topology is known. Unfortunately the topology is problem dependent and usually not easy to find. The genetic system presented in this study is an approach to solve this problem automatically. It consists of two innovative parts: the Semantic Changing Genetic Algorithm and the Unit–Cluster Model. The Semantic Changing Genetic Algorithm is an extended genetic algorithm that allows fast dynamic adaption of the genetic coding through population analysis. The Unit–Cluster Model is an approach to the construction of modular feedforward networks with a "backbone" structure.

Special Thanks

This special section is dedicated to all the people who contributed to this work and made it possible.

I would especially like to thank Prof. D.E. Rumelhart for inviting me to work with him at Stanford University. He greatly contributed to this study by active cooperation, providing me with his knowledge, his ideas and last but not least a comfortable working place. I really enjoyed working with him.

I would also like to thank Prof. E. Konrad for his interest in this project and his open mindness to me working at Stanford University.

Finally, I would like to thank my wife Małgorzata for encouraging and keeping up with me working, working and working. She took care of everything so that I was able to completely concentrate on this study.

Contents

Chapter 1

Introduction

Feedforward neural networks constitute a comfortable solution to many different problems. Unfortunately the selection of the correct parameters for the learning algorithm as well as the selection of the trainings data set and the topology[1] of the network is not a trivial task. This study focuses on the construction of problem specific topologies. Finding the correct topology is especially important as an unsuitable topology will keep the network from solving the problem. Either the network might not learn the problem or it will not be able to solve the generalization task. A topology with the smallest complexity[2] that learns the problem quickly and generalizes perfectly is optimal. It is however difficult, if not impossible, to find such a topology for non trivial problems manually. Genetic algorithms are a global optimization technique for high dimensional problems and can be used to solve this task.

1.1 Why Genetic Algorithms ?

It is assumed that there is no systematic way[3] to construct problem specific feedforward networks [34]. This means that there is no recipe for how to define the optimal topology of a neural network from looking at the trainings and test data sets[4]. Furthermore it is assumed that genetic algorithms are a more powerful approach to topology construction than constructive or destructive algorithms. Constructive algorithms start out with the empty (or a small) network and grow it by adding units and connections to the

[1]The topology of a neural network is defined by the amount of units of the network and their connectivity. The connectivity describes which units are connected to each other (it defines all connections which are not equal to zero), but *not* the strength of the connections.

[2]The complexity of a network is the amount of biases (units) and connections of the network.

[3]At least no systematic way has been found yet.

[4]For certain trivial problems this is possible, but with the growing complexity of problems that are supposed to be solved by applying neural networks, it becomes extremely difficult to find the topology manually.

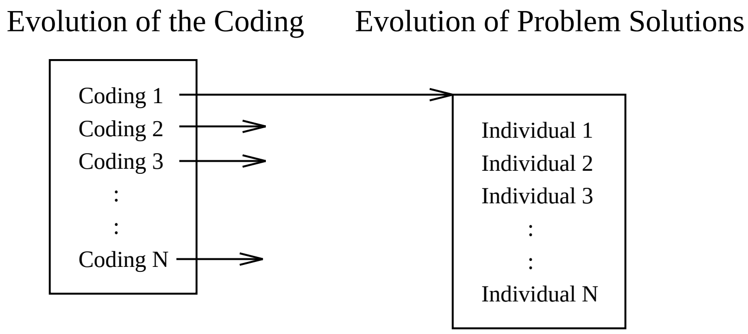

Figure 1.1: "Classic" evolution of genetic codings.

network according to certain growth rules[5]. Destructive algorithms, on the other hand, start out with a maximal network size and try to shrink the network by eliminating connections and units that do not contribute much to the solution of the problem. Both methods however are guided by a predefined heuristic, as it is computationally far to expensive to perform a "brute force" approach and try out all possible network structures. Genetic algorithms are *not* guided by a predefined heuristic, but are just guided by the raw performance of the networks and are thus able to cover the search space of topologies completely[6].

1.2 Goals

The goal of this research is to develop a genetic system that can be used for the topological design of feedforward neural networks. The network design is problem specific, meaning that for each problem an individual network must be evolved. The solutions should be at least sub–optimal topologies that can be trained with a learning algorithm – the backpropagation algorithm [29] has been used here – to solve the problem. A special emphasis is put on modular feedforward networks with multiple layers. In order to find modular neural networks two innovative approaches are employed – the semantic changing genetic algorithm and the restricted unit–cluster model. The semantic changing genetic algorithm is supposed to adjust the genetic coding, while searching for good topologies. It is unlikely that a perfect genetic coding can be designed manually and it is computationally to expensive to have a outer genetic algorithm search for an optimal coding as depicted in figure 1.1. The semantic changing genetic algorithm is expected to speed up the optimization process as it allows an interactive adaption of the genetic coding through population analysis, which is computationally relatively cheap. The unit–cluster model allows to create a genetic coding of feedforward net-

[5]E.g., they add units and connections whenever the network is not able to capture the problem after having been trained for a predefined period of time.

[6]All possible topologies of feedforward networks can be constructed.

works with a high emphasis on modular structures. The combination of both approaches results in a genetic system that is expected to perform well on problems that can be solved by modular feedforward networks. It should be noted that the system is *not* expected to be able to generate a solution for every possible problem (not a general problem solver).

Chapter 2

Neural Networks

This chapter gives a short introduction to neural networks. The biological motivation and the computational model are discussed first. Then the focus is directed to feedforward neural networks, as those are the networks, for which topologies will be genetically evolved. An argumentation is presented why modular neural networks might have advantages over the traditional completely connected networks. Furthermore a quick overview is given over the backpropagation algorithm that is used in this study to train the networks and measure their quality. Finally, it is discussed what important parameters may influence the ability of a network to learn and generalize a problem.

2.1 Biological Motivation

Conventional computers (von–Neumann computers) and the human brain solve tasks in a quite different way. Computers outperform the human brain, when it comes to precise mathematical calculations. The human brain, on the other hand, has abilities to e.g. recognize patterns (e.g. visual or acoustic) and coherencies and associate them, which by far exceed the capability of computers. It is therefore desirable to find a way how those advantages of the brain can be simulated on a computer. This would allow the computer to solve tasks at which it initially performs poorly. Those tasks, which include e.g. scene analysis and language and object recognition, are solved by the brain due to its massive parallel distributed knowledge representation and processing. It is suspected that more than 100 billions biological neurons are connected to each other in a highly complex way so that a large part of the brain can contribute to the processing of a signal [32]. Although the functionality of the brain is only known in its rudiments a simple model could be developed. The idea is to try to simulate the learning and the generalization abilities of the brain by developing artificial neural networks.

A neural network is traditionally a network of biological neurons that can be found in brains of humans and animals. Figure 2.1 shows a schematic

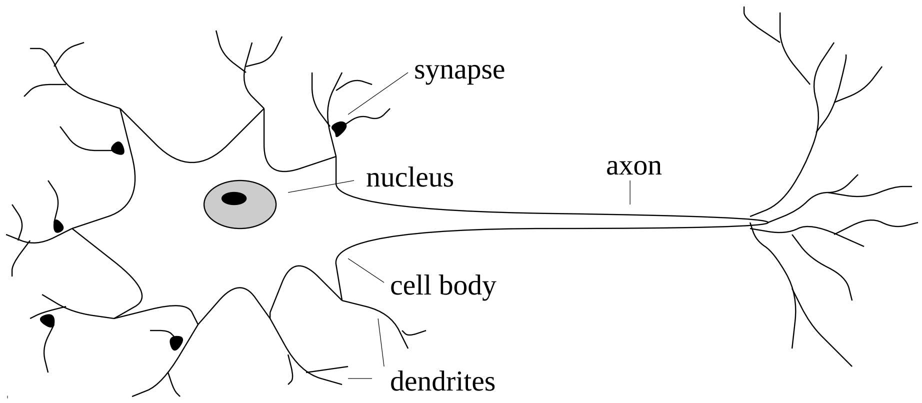

Figure 2.1: Schematic drawing of a biological neuron.

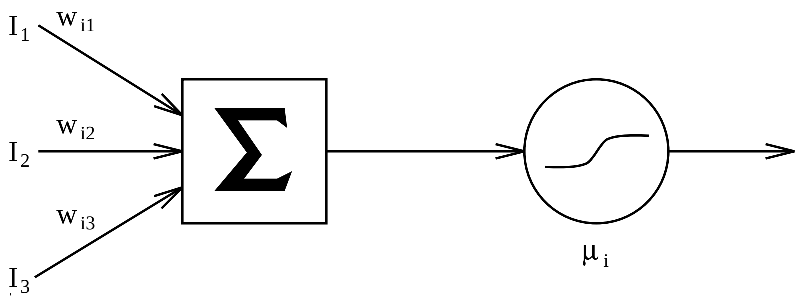

Figure 2.2: Schematic drawing of an artificial neuron.

drawing of a biological neuron. The neuron is connected to other neurons through dendrites, synapses and the axon. It receives stimuli through the dendrites, which react either inhibitory[1] or excitatory[2] to a given stimulus. When a certain minimal stimulus is reached the neuron "fires", meaning that it transmits a signal through the axon.

In order to build artificial neural networks a model of the neuron had to be established. Figure 2.2 shows a possible model. For the artificial neuron all its weighted inputs ($I_j \cdot W_{ij}$) are summed first. The weights simulate the behavior of the dendrites and act either inhibitory or excitatory to the input stimulus. The bias (μ_i) is also added to the sum. It can be regarded as a weight which always receives an input stimulus of 1. After the weighted and summed netinput (net_i) has been calculated it is applied to the activation function of the neuron. The activation function is often a non–linear, steady differentiable and monotone function. The function used in this study is the sigmoid function (logistic function) depicted in figure 2.3. The function output value is then used as an input stimulus for other artificial neurons in a network of neurons or is taken to be the result output value. Those artificial neurons, also called units, can be connected to each other to form

[1]diminishing the strength of the stimulus
[2]enlarging the strength of the stimulus

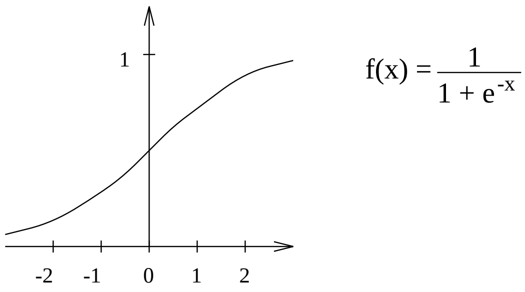

Figure 2.3: Schematic drawing of the sigmoid function.

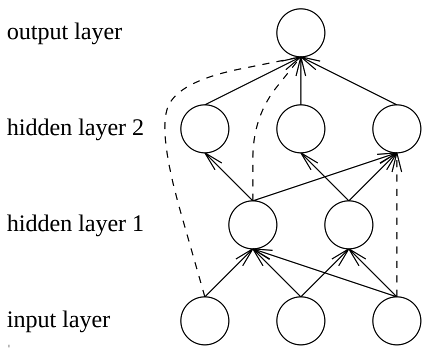

Figure 2.4: A multilayered feedforward network.

networks that are needed to solve complex problems which cannot be solved by a single neuron.

A detailed description of different models of neurons and a wide variety of neural networks can be found in [2, 7, 8, 26].

2.2 Feedforward Neural Networks

Feedforward neural networks are a special class of artificial neural networks that can only propagate an input stimulus in one direction through the network. An example of a multilayered feedforward network is shown in figure 2.4.

All units in such a network can be assigned to a certain layer. The networks may have three different kinds of layers – input, hidden and output layers. There is always exactly one input and one output layer. The input layer receives the input stimulus from outside and its only task is to propagate this input to the following layers[3]. The output layer, on the other hand, delivers the result after all input signals have been propagated through the network.

[3]The units of the input layer can be more primitive than the units presented in the previous section. There is not much use to weight the external input and put it through an activation function of a unit. Thus the only task of the units of the input layer is to distribute the incoming signal to units of higher layers.

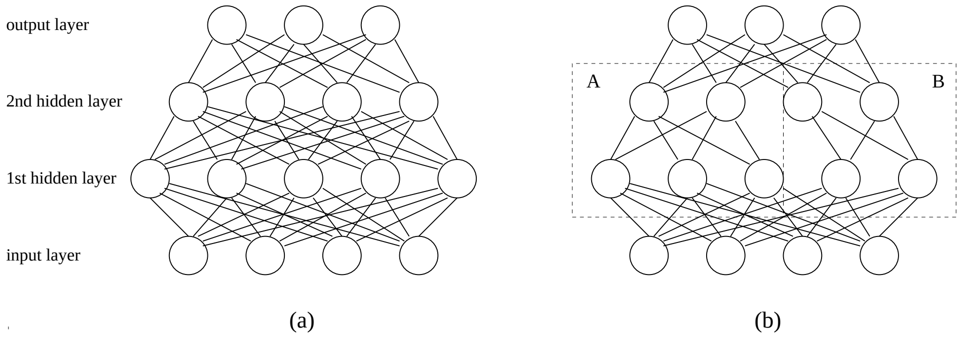

Figure 2.5: (a) A non–modular feedforward network. (b) A modular feedforward network.

All units that do not belong to either the input or the output layer are hidden units[4]. The hidden units can also be arranged in (hidden) layers. A unit is assigned to a layer through the following definition: *The layer of a unit can be calculated by regarding the longest existing path of connections*[5] *from the input layer to the unit.* Since there are only unidirectional connections in a feedforward network and no recursions or loops are allowed there is always a longest path from the input layer to each unit. The connections in figure 2.4 that are drawn with a solid line can belong to such a longest path. Those are connections that go from one layer to an adjacent layer. The connections drawn with a dashed line, on the other hand, skip layers and are called shortcuts.

All connections are weighted connections. After an input stimulus has been given to the input units, the signals are propagated through the network from unit to unit, with each unit acting exactly as described in the previous section, until they reach the output layer. Once all signals reach the output layer the result of the network can be read. The networks can be adjusted to solve different tasks by adjusting the weights of those connections. Sometimes however this is not sufficient to solve a problem and the topology of the network has to be changed.

2.3 Advantages of Modular Feedforward Networks

The class of feedforward networks can be divided into two sub–classes – modular and non–modular feedforward networks. An example for those two sub–classes is presented in figure 2.5. The network in figure 2.5 (a) is a non–modular network, as it contains no sub–networks. The adjacent layers of this network are completely connected to each other. The network in figure 2.5 (b), on the other hand, is a modular network, as it contains two

[4] Some feedforward networks do not have hidden units.

[5] A path of connections consists of consecutive connections.

independent sub–structures (A and B), which are *not* connected to each other. Modularity is therefore used here in the context of dividing networks into sub–networks that are expected to solve parts of the problem by itself.

Modular feedforward networks have some advantages over their non–modular counterparts [1] and a few of those advantages will be discussed here.

One major advantage is the lower complexity[6] of modular networks. If a modular network is able to capture a problem at all, it has the advantage that, due to its lower complexity, the search space[7] has fewer local minima and the desired solution can be found faster and more reliable when training it with a local (gradient descent) learn algorithm like backpropagation[8]. Furthermore, networks of smaller complexity are less likely to be affected by the overlearning effect[9].

Another advantage is the capability of modular networks to cope with interference problems. Interference problems can e.g. occur when a non–modular network is trained to learn two independent parts of a problem [27]. A network, which is originally large enough to capture the problem cannot be trained as it "forgets" one part of the problem while learning the other one. The network shows an oscillating behavior, but is not able to capture both problems at once. Modular networks often constitute a good solution to interference problems as they can assign one sub–network to learn one part of the problem and another sub–network to learn the other part.

An advantage of modular neural networks is also the limitation of equal solutions that can be found by such a network [1]. Modular neural networks are able to find less equally good solutions than their non–modular counterparts. This can be demonstrated on the example depicted in figure 2.5. Looking at the amount of permutations of units[10] in the two adjacent hidden layers leads to the following result:
In the non–modular network shown in figure 2.5 (a) a total of $4! \cdot 5! = 2880$ permutations are possible without changing the functionality of the network. In the modular network depicted in figure 2.5 (b), on the other hand, only $3! \cdot 2! + 2! \cdot 2! = 16$ possible permutations exist.
The solution becomes less ambiguous.

[6]The complexity of a network is defined here by the amount of connections and biases (units) and *not* by their arrangement in the topology.

[7]The search space of solutions is defined by all combinations of weight and bias values. The fewer weights and biases a network has the smaller is the search space.

[8]A large enough network will always learn the trainings data, but it does *not* have to find the *desired* solution. It is likely to find a solution, which will fit the trainings data, but perform poorly on the generalization task.

[9]Overlearning a problem means that after training the network the trainings data are approximated too well by the network. This leads to the effect that the test data are approximated worse and worse the longer the network is being trained. The network performs poorly on the generalization task.

[10]Permutations of units (and their connections) do not change the functionality of the network, if the permutated units have the same fanins and fanouts (are connected to the same units).

2.4 The Backpropagation Algorithm

There are several learn algorithms with which feedforward networks can be trained (their weights can be adjusted) to solve a problem. Some of them can e.g. be found in Hertz/Krough/Palmer [8]. This section gives a short overview over the backpropagation algorithm because it will be used in the evolution of feedforward networks presented in this study to train the networks and contribute to the calculation of their quality measure. A detailed description of the backpropagation algorithm can be found in [22, 28].

The backpropagation algorithms is a training algorithm for supervised learning of feedforward networks. A network is presented input and output patterns from an example training data set. Each input pattern in the set has its own desired output pattern. The patterns are presented to the network and the initially random weights of the network are adapted in an iterative process to make the network create for each input pattern the desired output pattern.

When an input pattern is presented to the input layer the network will propagate the input signals forward and an output pattern will be created at the output layer. In order to adapt the weights of a network and force it to create the desired output pattern, a difference between the desired output pattern (target) and the actual output pattern created by the network (output) has to be calculated. Usually this difference is measured by the total sum of squares (tss) over all output units:

$$tss = \frac{1}{2}\sum_i (t_i - o_i)^2$$

t_i: desired output of unit i
o_i: actual output of unit i

This sum constitutes the error function (also called fitness-, quality- or energy function) that has to be minimized for the network to learn. It is calculated for every pattern. There are two learning modes. Either the weight adaption is done after one pattern was presented to the network (online learning) or all patterns in the training data set are presented and then the weight adaption is performed (batch learning). Such a cycle is also called epoch. When the batch learning mode is chosen (as in this study) the tss is additionally summed over all patterns.

In order to adapt the weights so that the error function will be minimized the error (tss) has to be propagated back through the network. Partial derivations of the error function are calculated for all weights and the weights are changed in the direction of the resulting gradients. The weight changes from one epoch to the following one are calculated for each weight in the following way:

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot o_i$$

Δw_{ji}: weight change for the connection from unit i to unit j
η: learn rate

The learn rate η is user defined[11] and keeps the adaptive steps from being too large.
The δ_j–terms are calculated for output units as follows:

$$\delta_j = f'(net_j)(t_j - o_j)$$

and for hidden units in the following way:

$$\delta_j = f'(net_j)\sum_k \delta_k w_{kj}$$

with $f'(net_j)$ being the derivation of the sigmoid function ($f'(net_j) = f(net_j)(1 - f(net_j))$) and net_j the netinput of unit j.

The recursive formula allows to calculate the weight changes for every connection in the network. The weight changes are then added to the actual weights and one adaption cycle is finished. As the weights are adjusted for all dimensions in the direction of the gradient the backpropagation algorithm is also called a gradient descent algorithm. The advantage of the gradient descent technique is that a minimum is found quickly, the disadvantage, on the other hand, that often only a local minimum can be reached. Gradient descent techniques belong therefore to so called local search strategies.

The backpropagation algorithm used in this study is extended by a momentum term. The momentum term rates the gradient of the previous cycle. The backpropagation algorithm takes this gradient into consideration when calculating the weight change (see [22] for a detailed explanation). The momentum term is not needed for the algorithm to work, but it might speed up the finding of minimum significantly.

2.5 When do networks capture problems?

In this section several parameters are mentioned that take strong influence on whether a feedforward neural network will find a solution or not. It is meant to give some insight to the general behavior of the networks. The parameters include the topology of the network, the random initial weight initialization, the learn parameters of the backpropagation algorithm and the train and test pattern data sets.

[11] There are extensions to the backpropagation algorithm that adapt this parameter automatically [30].

2.5.1 Topology

A well suited topology is highly important for the network to find a solution. Networks which are too small are not able to learn the trainings data set at all or they will just learn it partly. Networks which are too large, on the other hand, can keep the network from generalizing after it has been trained. However, not only the size of the network is crucial to whether it will capture a problem. Networks of equal size may and may not be able to constitute a solution depending on their topology. The optimal topology of a network is the one with the smallest complexity which still allows the network to learn the trainings data and generalize the test data. Such a topology however can be quite difficult to find.

2.5.2 Weight Initialization

The influence of the weight initialization can be illustrated by looking at two extreme situations:

1. If the given weights of the network are set (random or on purpose) in such a way that an adequate solution to the problem is found, then the network does not have to be trained any further.

2. If the weights of the network constitute a solution which lies in a local minimum, then a local search technique like the backpropagation algorithm might not be able to improve this solution.

Besides those extremes it can also happen that with different (random) weight initializations a network might once find a solution, solve the problem just partly (get stuck in a local minimum) or not at all[12]. It is therefore sometimes necessary to conduct several training runs with different weight initializations in order to reach a solution. The size of the interval from which initial values are selected during a random weight initialization takes also influence on the learn behavior of a network. Good values have been established to lie in the interval [-0.5..0.5], which belongs to the dynamic part of the sigmoid function. The size of this interval can be decreased in order to make the random weight initialization less influential, but may result in a longer learning phase for a solution to be reached.

2.5.3 Learn Parameters

Different learn algorithms have different learn parameters that have to be set, but since the backpropagation algorithm is used in this study just the most crucial parameters of this algorithm will be mentioned here.

[12] A good example is the standard solution for the XOR problem. This network learns the XOR in about 66% of all training runs.

The most crucial parameter of the backpropagation algorithm is the learn rate. This parameter influences the convergence behavior and the duration of the training phase. A secure convergence (at least to the next local minimum) can be achieved by setting the learn rate to an extremely small value, but this usually also leads to a long training phase that is needed in order to reach the solution. Setting the learn rate to a larger value may speed up the convergence to a certain point, which is problem dependent, but might also keep the network from learning at all.
Salomon [30] e.g. presents a technique how the learn rate can be adapted automatically. The idea is to adapt the learn rate in relation to the roughness of the surface of the error function. When an area on the surface is reached where the surface is rather rough the learn rate is kept small, while when an area is reached that is smooth the learn rate is enlarged in order to speed up the learning progress.

Another important parameter is the momentum. A large momentum value usually speeds up convergence, but might be disastrous when rapid changes in direction during gradient descent are needed to improve the error of the network.

2.5.4 Train and Test Patterns

The selection of appropriate training and test data sets is highly important for obtaining valid results. Usually a neural network is not used to just memorize the trainings data, but it is supposed to be able to generalize once it has been trained with the trainings data. In order to check the generalization capability of a network a second data set is applied – the test data set. Both data sets have to be created carefully. The trainings data set is supposed to be representative, meaning that it should comprise the entire problem. Furthermore, it is important to make sure that no other generalization is possible than the one described by the test data set[13].

When using the online learning method, where the weights are updated for each pattern presented, the order in which the patterns are presented might be important. It is advantageous to change the order from time to time in order to prevent lethal cycles during which the achievements of an adaption after a pattern has been presented are undone by the following adaption caused by the pattern presented next.

[13] A humorous example of how *not* to create a trainings data set has been presented by the US Department of Defense. The goal was to train a network to classify whether an image contained a tank or not. The network learned extremely well, but unfortunately only to distinguish whether the sun was shining or not due to a wrongly designed trainings data set. All images with tanks in them were shot on a sunny day and all images without a tank on a cloudy day. The network learned the difference in brightness.

2.6 Conclusion

This chapter gave a short introduction to artificial neural networks. The motivation for creating artificial neurons was founded in the differences between the abilities of common computers and the human brain.

The functionality of biological neurons was described. The level of abstraction chosen to create a model of a neuron distinguished between the following parts of a neuron:

- synapses
- dendrites
- cell body
- nucleus
- axon

The established parts of a biological neuron were then modeled in an artificial neuron as follows:

- connection weights (synapses)
- connections (dendrites)
- sum function (cell body)
- activation function and bias (nucleus)
- output (axon)

The focus was directed to a special class of artificial neural networks that can be build with those artificial neurons – feedforward networks. A definition was given that stated that feedforward networks are just allowed to propagate input signals unidirectional. Their layered structure, which includes input, hidden and output layers was explained.

The class of feedforward networks was divided into modular and non–modular feedforward networks. A comparison of those two classes was made and the following advantages of modular feedforward networks were elaborated:

- lower complexity
- more suitable for interference problems
- limitation of equal solutions

The backpropagation algorithm was shortly presented. Its main idea and mathematical model were described. The backpropagation algorithm was classified to be a gradient descent search algorithm that is able to quickly find a local solution by iteratively adapting the weights of a feedforward network.

Finally, the influence of several parameters on the learning and generalization capabilities of networks was described. Those parameters were:

- topology
- weight initialization
- learn parameters
- train and test patterns

The analysis showed that there are many parameters that have to be set in an appropriate and problem dependent way for networks to be able to solve problems.

Chapter 3

Genetic Algorithms

This chapter first gives a short introduction to genetic algorithms (for a detailed description see Goldberg [10]). Then the importance of the genetic coding is emphasized and criteria are established that can be used to measure the quality of a genetic coding. Furthermore an overview is given over the simple genetic algorithm as it was introduced by Holland [9]. Then the three different variations of the simple genetic algorithm used in this research are presented. The variations include the extended simple genetic algorithm (ESGA), the semantic changing genetic algorithm (SCGA) and the extended semantic changing genetic algorithm (ESCGA). First, the ESGA is described. Then the need for a semantic changing extension is explained and two solutions – SCGA and ESCGA – are introduced and discussed. Finally test results are presented in order to show some capabilities and limits of the newly introduced algorithms.

3.1 Introduction to Genetic Algorithms

Genetic algorithms are a global optimization technique for high dimensional, non–linear and noisy problems. The success of genetic algorithms is founded in their ability to keep existing parts of a solution, which have a positive influence on the outcome, and proceed with optimizing the non–optimal parts. The approach is global as many samples from different parts of the solution space are examined simultaneously and as the transition rules, which combine and change those samples in order to improve the solutions already found, are probabilistic and not deterministic. This enables genetic algorithms to reach a global optimum without getting stuck in local optima.

In order to solve a problem using genetic algorithms the parameters of the problem (phenotype) must be coded onto a genotype. A genotype is often represented by a bit string, an integer- or float number string or any other symbol string (trees are also possible [14]). The genetic algorithm works on a population of genotypes searching globally for the optimum by applying genetic operators (see section 3.4) to the genotypes with respect to their quality. The quality of a genotype is calculated by extracting the phenotype it represents and evaluating the fitness of the phenotype. In the case of topology optimization this means that a symbol string (genotype) can be decoded into a network (phenotype). The fitness of the network is then calculated with respect to a quality function that may include the ability of the network to learn (train error) and generalize (test error), its complexity (amount of units and links) and various other criteria.

3.2 Genetic Coding

The choice of the genetic representation is crucial to the performance of the genetic algorithm, as the genetic coding defines the window through which the genetic algorithm sees the world [15]. The power of a genetic algorithm to solve a specific problem very much depends on *how* and *in which order* the parameters of the phenotype are coded on the genotype [10, 17].

Several criteria can be found to decide, whether a genetic coding is efficient and promising:

- **completeness**
  A genetic coding is considered complete if every point in the search space can be encoded [11].
  This ensures that all solutions are accessible by the genetic algorithm. If e.g. the task is to find the topology of a feedforward neural network, it should be possible to encode all possible feedforward neural networks. Ignoring this criterion could result in loss of possibly valuable solutions.

- **closure**
  A genetic coding is closed, if all individuals produced by applying genetic operators can be decoded to valid phenotypes [11] (provided that their parents have been valid individuals).
  If this criterion is met, the search space can be kept minimal. No points, which do not represent a solution, can be accessed by the genetic algorithm. This keeps the genetic algorithm from wasting time on searching through parts of an enlarged search space that definitely do not contain valid solutions. In the example of network construction meeting this criterion would ensure that all genotypes can be decoded to valid feedforward networks[1].

- **proximity**
  A genetic coding meets the criterion of proximity, if a gradual improvement of the parameters is assured.

[1] Networks e.g. without any connections from input to output layer (with no path going from input to output layer) could *not* be encoded.

It should be possible to adjust the coded parameters gradually in a monotone way so that little changes to a parameter's genetic coding should just have a small influence on the solution, while major changes should influence the outcome in a great way.
In the example of network encoding this states that a small change in the coding of a network should change its topology in a very minor way (e.g. adding or subtracting a connection/unit), while a great change in the coding should result in a quite different topology (e.g. many more/less units/links or a completely different connectivity).
A further example where this criterion is met is given by gray coding of numbers. Gray coding of numbers ensures that numbers that are mathematically closest to each other are coded binary with a minimal bit difference. A counterexample is the plain binary coding where two adjacent numbers may differ in almost all their bits (e.g. 100000 and 011111).
A work displaying the influence of proximity on genetic coding of neural networks has been done by Marti [17]. He showed that lack of proximity may cause the genetic algorithm to encounter great difficulties in solving even simple problems[2].

- **short schemata**
  The criterion of short schemata is met, if parameters, which are related close to each other, are coded close to each other on the genostring. Parameters are related close to each other, if they solve a part of the problem together.
  This criterion cannot always be assured, as it sometimes requires a deep insight into the problem. Nevertheless it is of great advantage to put those building blocks close together on the genotype. A detailed description of the schemata theory[3] can be found in [9].

- **compactness**
  A genetic coding is compact, if the length of the genotype is minimal

[2] Marti encoded three different kinds of connections binary – inhibitory, non–existent and excitatory – in two different ways. First he encoded inhibitory connections with 10, non–existent connections with 00 and 01 and excitatory connections with 11. Using this encoding scheme the genetic algorithm was handicapped. In order to change a non-existent connection (coded with 00) to an excitatory one the genetic algorithm had to jump over a state of even lower fitness (an inhibitory link coded with 10) to reach the improved state (the excitatory link coded with 11). Marti found out that such a coding greatly disturbed the expected convergence behavior. After coding the connections in a different way the genetic algorithm converged faster and more reliable. The second coding set the inhibitory connections to be represented by 00, the non–existent by 01 and 10 and the excitatory by 11. Thus a gradual and monotone improvement was possible.

[3] Parameters, which are encoded close to each other on the genotype are less likely to be separated (changed) by the application of crossover than parameters, which are encoded far apart from each other (with a two point crossover the genostring must be regarded as being circular in such a way that the end is adjacent to the beginning). Thus it is reasonable to encode parameters that constitute a unity (work on the same part of the problem) close to each other.

[11].
Compactness can also be used as a measure. The shorter (composed of less parts) the genetic coding of a solution (genotype) is the fewer steps the genetic algorithm has to go through in order to find a solution and thus the faster a solution is reached.

- **non–isomorphism**
  A genetic coding fits the criterion of non–isomorphism, if each genotype represents a truly unique solution (phenotype).
  The aspect of avoiding isomorphism is already partly covered by the compactness criterion. If many different genotypes are decoded to the same phenotype, then the coding is not compact. Furthermore the chance of creating poor offsprings is increased as combining two successful individuals, which have a completely different genetic coding may easily lead to the loss of valuable information in both of them[4].

- **modularity**
  A genetic coding is modular, when partial solutions (modules/building blocks) are written out just once and a mechanism is provided so that they can be referenced at different points in the coding [11].
  It is much like in a program where a subroutine is defined just once and then referenced many times throughout the program. Modularity is necessary to ensure compactness.

This list does not claim to be complete as more criteria can probably be found. Nevertheless the criteria presented here can be viewed upon as a minimum requirement for a genetic coding to be efficient and promising.

3.3 Simple Genetic Algorithm

The original genetic algorithm introduced by Holland [9] can be written down in the following way:

[4]Let's say we have a coding that allows one to define a feedforward network that consists of two serially chained networks – a XOR–net and an AND–net – in two different ways:

genotype	decodes to	phenotype
XOR–net,AND–net	$\rightarrow$	XOR–net – AND–net
AND–net,XOR–net	$\rightarrow$	XOR–net – AND–net

Combining those two individuals (let's assume they are almost perfect solutions) using a one–point crossover could result in:

XOR–net – XOR–net
and
AND–net – AND–net

Thus the previous structure would be lost completely.

1. Create random initial population
2. for each generation do
   (a) selection
   (b) production
   (c) reduction
3. end for

After creating an initial population of random individuals, the evolutionary loop begins with the evaluation all individuals. Then individuals are selected for reproduction with respect to their quality. During the production phase different genetic operators are applied to the selected individuals with a certain probability. After the production phase the population contains individuals from two generations – parents and offsprings. During the reduction phase a decision is made, which individuals advance to the next generation and which are discarded. The next generation begins then again with the evaluation of the population.

3.4 Genetic Operators

In this section a short explanation of the genetic operators: selection, crossover and mutation is given and the reduction strategy used in this research is presented. For further and more detailed information on genetic operators and different reduction strategies see [10]. In order to show the properties of the genetic operators used in this research the genotype is chosen to be a bit string.

3.4.1 Selection

The process of selection ensures that individuals of higher quality are more likely to be chosen for reproduction than those of lower quality. There are several ways of selecting individuals and **roulette wheel selection** has been used in this research.

Roulette wheel selection is a special case of *fitness proportionate selection*. Individuals are chosen for reproduction with probabilities directly proportional to their quality. The following example should visualize that procedure:

If there is a population of four individuals with the following qualities:

individual		quality
1	$\rightarrow$	8
2	$\rightarrow$	4
3	$\rightarrow$	2
4	$\rightarrow$	2

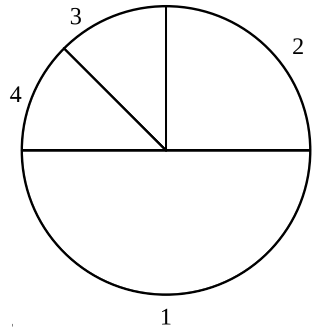

Figure 3.1: **roulette wheel selection:** individual 1 will be selected for reproduction with a probability of 50%

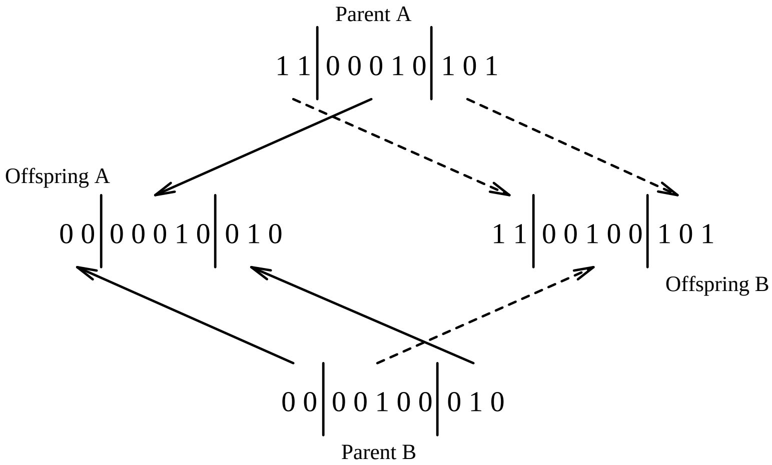

Figure 3.2: **two point crossover**

then individual 1 has a probability of 50% to be selected for reproduction each time an individual is selected (each time the wheel is spun[5]). In order to find the probability for each individual one has to calculate the sum of qualities of all individuals and divide the quality of each individual by this sum. In the example for individual 1 this would be 8 divided by (8+4+2+2 = 16) equals 0.5. Figure 3.1 shows how the wheel is divided for the example population.

3.4.2 Crossover

The crossover operator combines two previously selected individuals as shown in figure 3.2 and yields two offsprings.

This operator tries to combine vital parts of two individuals in order to create a superior individual. Although it is possible to choose a crossover operator with a different amount of crossover points, a two point crossover

[5]The wheel is usually spun once for each offspring that will be created.

1 1 0 0 0 1 0 1 0 1

↓

1 1 0 1 0 1 0 1 0 1

Figure 3.3: **mutation**

has been selected, as crossovers with multiple points do not seem to improve the optimization process [10]. During the crossover operation two points in the symbol strings (genotypes) are arbitrarily chosen and the part, which is enclosed by those two points is swapped. Individuals selected for reproduction undergo crossover with a certain probability or advance unchanged to the offspring population.

3.4.3 Mutation

Mutation, which is the main engine of the traditional evolution strategy [25], is used with genetic algorithms to introduce new solutions and prevent the population from unrecoverable loss of important information. Mutation is accomplished by modifying single symbols of the symbol string with a certain probability. In the example of a bit string coding (see figure 3.3) this is done by flipping bits.

The probability for mutation is usually kept low to prevent a negative influence on the crossover operation and in order to maintain a strong causality. With a high mutation rate (e.g. 0.5) the genetic algorithm would turn into a random search.

While crossover roughly establishes the region of the search space, which contains the solution, mutation is additionally useful for “fine tuning” at the end of the optimization.

3.4.4 Reduction

At this point the interim population contains parents and offsprings (some offsprings may be identical with their parents). During reduction individuals are selected for entering the new generation. The reduction plan in this research states that all but one individuals of the parent generation (or 90%[6] are replaced by individuals of the offspring generation. The *elitist* strategy is used so that the best individual of a given generation automatically advances to the next generation.

[6] see section 4.3.1

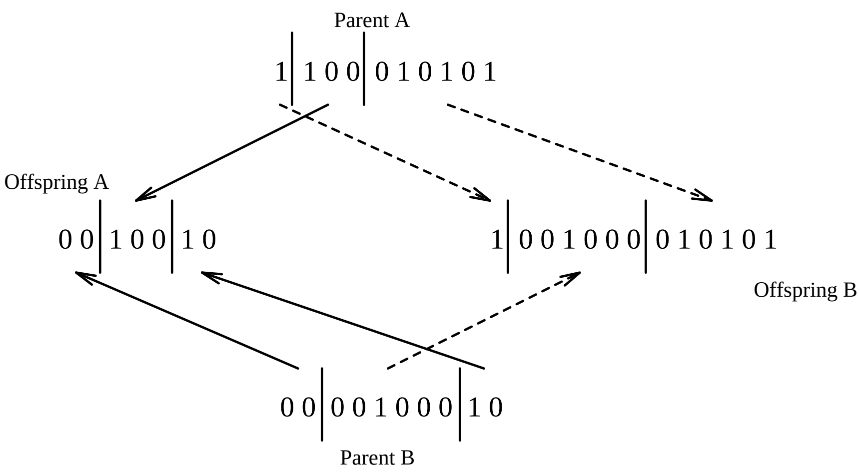

Figure 3.4: **two $\times$ two point crossover**

3.5 Extended Simple Genetic Algorithm

The SGA does not present a good solution for some problems due to its fixed length genotypes. Optimizing topologies is one of those problems, as the size (amount of hidden units) and complexity (amount of links and biases) of the network is unknown in the beginning. Thus it is hard to define arbitrary networks ensuring a compact coding and using the SGA. The extended simple genetic algorithm (ESGA) is a simple genetic algorithm, which also operates on a population of bit strings (or any other symbol strings), but in contrast to the SGA the genotypes may vary in length.

The initial population is randomly created and all individuals (bit strings) have the same length. During crossover the coding of an individual may grow or shrink. Two crossover points are selected *for each* individual and again the parts enclosed by the two points are swapped creating two offsprings. This time however it is possible that the substrings, which are swapped, have a different length. That enables the creation of individuals, which differ in length. Figure 3.4 shows this procedure.

3.6 The need for an extension to the genetic algorithm

The previously described genetic algorithms are powerful optimization tools for many applications. Nevertheless, when it comes to the construction of topologies of neural networks they lack some features, which could be useful. Some of those features are pointed out in this section:

- **controlled growth of the genetic code**
  If the size of the genotype is not known – like in most genetic codings

of neural networks – the ESGA would be used usually as it can adapt the length of the genetic coding automatically. But with the ESGA the only influence the user can take on the speed with which the genotypes grow is through setting the initial size of the individuals and defining the fitness function. This might be a limitation, which is too tight. It might be advantageous to search for an optimum for each size of a genotype for a while before enlarging the individuals[7], but this is definitely *not* a feature of the ESGA as test results will show.

- **automatic module definition**
  This feature would allow modules (sub–networks) to be created once and referenced many times without redefinition. The ESGA–crossover may distribute a substring to many individuals and various locations, but a copy of the substring (redefinition) will then be present at all the locations and not just a reference. That references can be used successfully has been shown in the works of Koza [14] and Gruau [11].

- **extended module distribution**
  Once a valuable module is found it should be possible to distribute it quickly to individuals of lower fitness. The idea is very simple and introduces a technique, which is found in society and in natural evolution. It postulates that once a valuable module has been found by one individual it should be quickly (in the same generation) passed on to individuals, which perform poorly so that they can use it to improve their fitness. It is much like ideas in society spread and like plasmids exchange genetic material[8] [16]. This feature should be tested, whether it really has a positive influence as it might force the population to converge to a local minimum. It is a feature which is unknown to the SGA and the ESGA.

- **module isolation**
  This point is just of minor importance as it does not necessarily contribute to the functionality of the extensions.
  Sometimes it might be interesting to see which modules are valuable. One would have to isolate the modules in the genotype. This also can be done automatically.

- **automatically meeting the proximity criterion**
  When establishing a genetic coding (e.g. for neural networks) it may be very difficult to design the genetic coding in such a way that the

[7] If e.g. the size of a network is proportional to the size of the genotype, it would be useful to know, whether there is a network of a small size that solves the problem before trying out larger networks.

[8] Plasmids are small bacteria that are able to exude parts of their DNA. Those parts float (outside the body of the bacteria) until their are picked up by another bacteria or virus not necessary of the same kind. Those bacteria or viruses that pick up floating genetic material incorporate it into their own DNA. This may lead to situations where viruses become immune to drugs they have never been treated with.

proximity criterion is met. Thus it would be advantageous to have a feature that would enable the genetic algorithm to find the right coding by itself (given just an alphabet of all parameters that should be included).
Marti [17] showed that this is a very important issue. He used a very simple example where it was easy to find the correct coding manually, but this is often not the case[9].

- **supporting closure**
  There are several ways of ensuring the closure criterion. It is possible to create a genetic coding so that this criterion is met[10] and thus an extension to the ESGA (or SGA) is not need. Another option is to limit the crossover points to be of the same kind [14, 15]. This means that once a crossover point is selected the matching crossover point of the mating partner must be found in such a way that the offspring will be a valid individual. Similar thoughts should then be given to the mutation operator.

- **integrating knowledge**
  Specific knowledge of the problem or parts of it, if made available to the genetic algorithm, can speed up the optimization significantly. If e.g. modules that solve a part of the problem are known they should be given to the genetic algorithm in form of a priori knowledge. The only possible ways to include this kind of knowledge using the ESGA is either by designing initial individuals that contain this information or by changing the genetic coding. Both approaches have their disadvantages. Including knowledge by designing the initial population may result in loss of that knowledge at an early stage of the optimization[11] or extinguish all randomness in the population[12]. Enlargement of the genetic coding in order to include the knowledge puts one back to the problem of meeting the proximity criterion[13], but is a possible solution.
  Thus there seems to be a need for an extension which would provide a possibility to include such knowledge automatically ensuring that this knowledge will be available to the genetic algorithm throughout the entire optimization[14].

[9] See e.g. the coding of neural networks used in this study.
[10] This e.g. is the case, if every arbitrary genotype can be converted into a vital phenotype.
[11] Imagine that some other part must develop first for the a priori given module to take influence on the fitness. In this case individuals that contain the module could disappear before the module comes into effect.
[12] If a part of the initial population contains designed individuals and another part contains random individuals, then the designed individuals are likely to take over the entire population in just a few generations (provided they are of higher fitness than the random ones).
[13] The question arises how to encode a complex module.
[14] by e.g. including it in the genetic coding in a way that the proximity criterion is met

Again this list is not necessary complete as more features could probably be found, but it contains some useful features that should be included into an extension of the genetic algorithm.

3.7 Semantic Changing Genetic Algorithm

The semantic changing genetic algorithm (SCGA) contains an extension to the simple genetic algorithm, which enables most of the features described in the previous section. The SGA is given an add on extension that directly manipulates the genetic coding of the parameters (of the phenotype). Two stages of genetic coding are used – a higher and a lower level genetic coding. The meaning of the lower level coding is being dynamically adjusted. While the optimization of the problem is performed on the lower level coding the dynamics of the higher level coding are used to determine the proper coding, to find vital modules and distribute them quickly, to control the growth of the individuals and to enable the integration of a priori knowledge.

The main frame for the SCGA is the SGA on bit strings. The extension introduces a second level genetic coding. The main idea is to translate the bit strings into meaningful symbol strings (or rules) using a self adapting symbol table. The symbol strings can then be decoded to phenotypes (e.g. neural networks).
The symbol table is self adapting in such a way that it captures sub–symbol strings, which probably make a positive contribution to the solution of the problem, thus making those sub–symbol strings available to the entire population.

3.7.1 Initial Settings

In order to begin the optimization three *major* definitions have to be made:

1. An alphabet of axioms and its distribution percentage in the initial table must be defined first. The alphabet should contain at least all the elementary parameters/parts/functions needed to encode a desired solution. It is also possible to include more complex functions (modules), if those are known. The distribution percentage of the letters of the alphabet defines their occurrence in the initial symbol table. If nothing about the occurrence is known, they should be distributed equally.

2. Then the size of the translation table must be specified. To do so one has to define how many bits will be used to address one translation table entry. It is an indirect specification that allows table sizes of 2^n, where n is the amount of bits. The minimum table size is the size of the alphabet. If a larger table size is chosen, the letters of the alphabet will occur more than once in the initial table. The size of the symbol

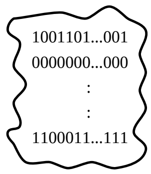

Figure 3.5: An example of a random population of bit strings

table defines the amount of different modules, which can be captured by the algorithm.

3. Finally the total amount of bits for the initial[15] individuals has to be set (the length of the initial bit string genotypes)[16]. It defines the granularity of the symbol string. The longer the bit string is the more parts (different modules from the symbol table) the symbol string will have.

Of course, one also has to set all the parameters, which are required by the SGA that builds the frame for the SCGA. There are some additional SCGA specific parameters that have to be set at the beginning of the evolution, but they will be explained later, when their usage will be described.

3.7.2 Initialization

After defining the basics described in the previous section the optimization can begin. The first phase of the optimization is the initialization phase. During this phase two structures are created:

1. *The initial bit string population.*
   The size of the population as well as the length of the bit strings are user defined[17]. The bit strings are created randomly (see figure 3.5).

2. *The initial translation table.*
   The initial translation table is created using the alphabet, its percentages and a fixed table size. The table contains 2^n table spaces where each table space is indexed by n bits. The letters of the alphabet are placed randomly in the translation table (see figure 3.6).
   First, all the letters which are given an explicit percentage for their occurrence are placed in the table and then all the remaining letters that should be distributed equally are randomly inserted. The letters are inserted on a first come first serve basis. This means that letters, which occur earlier in the alphabet have a chance of occurring once

[15] The amount of bits of an individual may change during evolution with the *extended* semantic changing genetic algorithm, but *not* with the SCGA.

[16] This parameter should be set to a multiple value of the address length of one table entry defined previously.

[17] All bit strings have the same length (amount of bits).

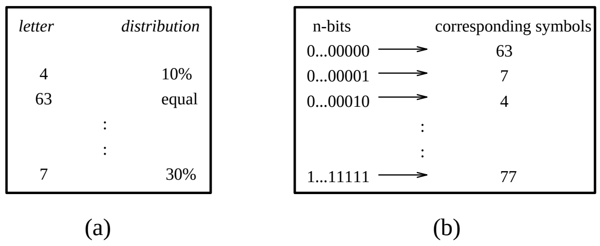

Figure 3.6: (a) A possible initial alphabet. (b) An example of an initial translation table that uses the alphabet. All table spaces are initially occupied by exactly one letter of the alphabet.

more in the translation table than letters at the end of the alphabet. If it is not possible to insert all letters in such a way that all table spaces are occupied and all letters are distributed, the percentages should be adjusted so that they sum up to 100% or the translation table must be enlarged.

After the initialization of the evolutionary process its main loop begins.

3.7.3 The Genetic Loop

The main genetic loop is depicted in figure 3.7. The loop starts with a population of bit strings and contains the following steps:

1. **translate bit strings to symbol strings**
   This is the beginning of the add on extension to the SGA. During this step the lower level genetic representation (bit strings) is translated to the higher level genetic representation (symbol strings) as will be explained in section 3.7.4.

   The result is a population of bit strings together with their corresponding symbol strings. It is necessary to keep both representations as future steps will show.

2. **calculate fitness**
   At this point the fitness of each individual can be established. In order to calculate the fitness, each symbol string (individual) is decoded to the phenotype it represents, the fitness of the phenotype is established and assigned to the genotype[18]. As the fitness calculation is very problem specific, it is not further discussed here. Some examples will be given with the tests described in this chapter and an appropriate fitness function will be introduced for evolving feedforward neural networks (see chapter 4).

[18] In the example of neural network construction this would mean that the symbol strings are decoded into neural networks and those networks are then tested for their performance.

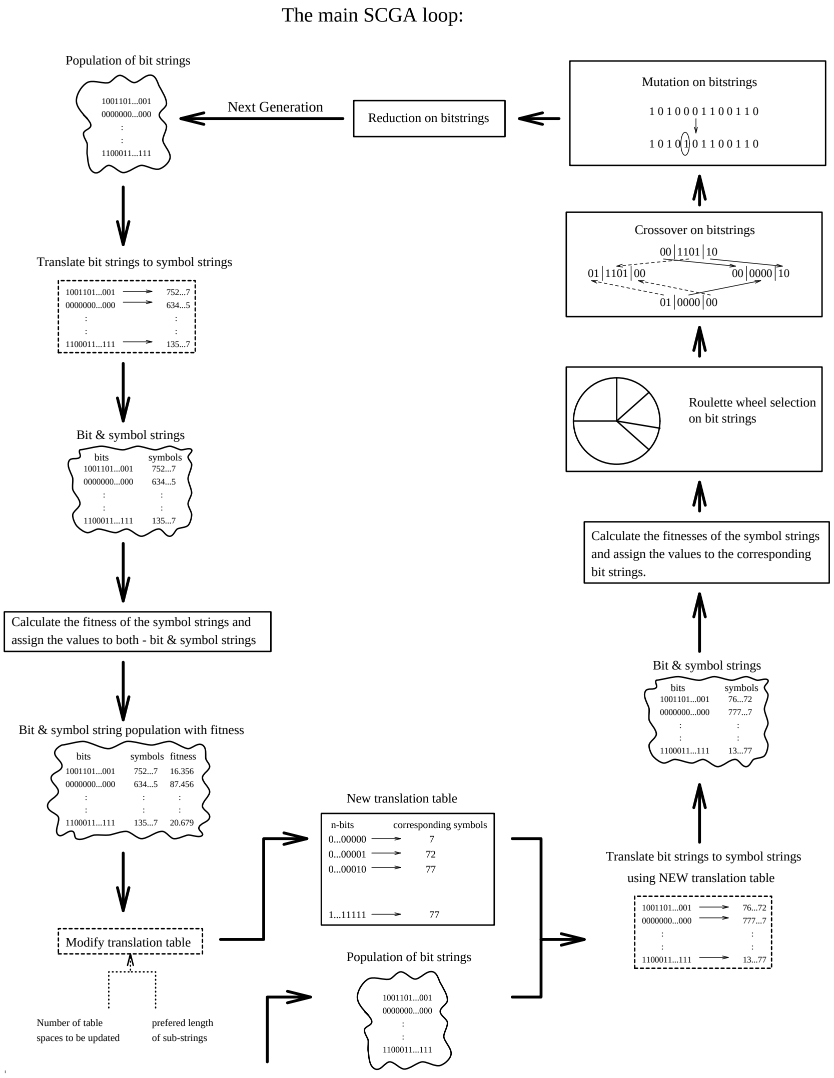

Figure 3.7: The main genetic loop of the SCGA.

After this step each bit- and corresponding symbol string in the population have an assigned fitness value[19].

3. **modify translation table**
   This is actually the semantic changing part of the SCGA. During this phase the translation table is modified. The modification is a complex procedure and therefore described in a special section (see 3.7.5). This main part of the add on extension to the SGA enables most of the

[19] In theory it does not matter whether the best fitness value is 0 or any other number. In this explanation of the functionality of the SCGA it is assumed that the higher the fitness value the better the individual. It seemed to be the easiest way of describing the functionality. Nevertheless the implemented algorithm used for testing is designed in such a way that the optimal quality is 0 and all qualities are only allowed to be equal or greater then 0. The smaller the fitness value is the better is the individual.

features described in section 3.6.

The result of this step is a new translation table. With a new translation table the symbol strings and therefore the qualities do not necessary correspond to the bit strings anymore. This brings us back to a state quite similar to the beginning of the loop, where there was just a bit string population (and a translation table). This step also marks the end of the add on extension to the SGA.

4. **translate bit strings to symbol strings**
   This step is equal to the first step in the loop, with the only difference that at this point the translation table has changed and therefore different symbol strings are derived from the bit strings.

   It again yields a population of bit strings and corresponding symbol strings.

5. **calculate fitness**
   The fitnesses are calculated again using phenotypes decoded from the symbol strings and are associated with the corresponding bit strings[20].

   The result of this step is an evaluated bit string population.

6. **selection**
   From now on the loop continues exactly as if this were a SGA. *Roulette wheel selection*[21] is executed on the bit string population.

   The amount of selected individuals is equal to the size of the original population.

7. **crossover**
   A two point crossover is applied to the selected bit strings (individuals) with a user defined probability. With a user defined probability of less than 1.0 some individuals are likely to advance without being affected by crossover.

8. **mutation**
   At this point the mutation operator is applied to the selected bit strings (individuals). The user defined mutation rate sets the probability with which a bit is flipped (for each bit in each individual).

9. **reduction**
   This is the last step of the loop. The elitist strategy is used and thus all but one individuals[22] of the parent generation are replaced by their

[20] It is often (except on very noisy problems, where one wants to evaluate an individual over and over again) sufficient to just evaluate the individuals which actually have changed due to table modification. This can cut down the amount of fitness calculations significantly Especially when just small portions of the table have changed and the the bit strings access only few table entries.

[21] Any other selection method could be chosen as well.

[22] 90% in the tests for topology evolution. 10% are randomly taken from the parent population.

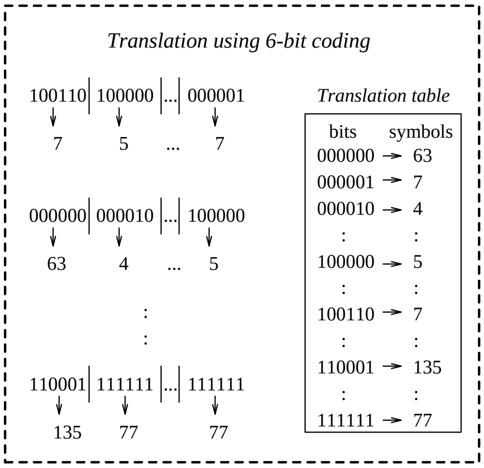

Figure 3.8: Translation of the lower level genetic representation to the higher level genetic representation.

offsprings[23]. Additionally the best individual of the parent generation is copied unchanged to the new generation.

After reduction a population that contains unevaluated bit strings is given and the next generation begins[24].

3.7.4 Translation of Bits to Symbols

In order to calculate the fitness of the bit string population and in order to modify the translation table the bit strings (lower level genetic representation) have to be translated into symbol strings (higher level genetic representation). The translation is a very simple straight forward procedure as depicted in figure 3.8.

The bit string is read from left to right and n bits that address one table entry are read at a time. Those n bits are then translated into one entry of the symbol table[25]. The result of translating all those addresses is a symbol string. The size n was initially defined by the user (see section 3.7.1). If at the end of the bit string some bits remain that cannot be translated into a table entry as their amount is smaller than n, they are ignored.

The procedure of translation yields a population of symbol strings. The translation is unidirectional and only possible from bit- to symbol strings.

[23] Offsprings are all individuals who have been selected for reproduction and passed the reproduction (crossover and mutation) phase. Some offsprings may be identical with their parents.

[24] During the evolution of topologies all individuals are reevaluated due to the noise in the problem introduced by the random weight initialization of the networks. If the problem does *not* contain any noise, individuals that have not been changed during crossover and mutation do not need to be reevaluated.

[25] An entry in the symbol may actually consist of several symbols of the alphabet. The symbol combination e.g. may describe a module.

The symbols often cannot be translated back into bit strings as the translation table may include some ambiguity (two different bit strings may be translated into the same symbol string).

3.7.5 Automatic Table Adaptation

The automatic table adaptation is the key mechanism of the semantic changing extension to the SGA. This extension should support most of the features mentioned in section 3.6. To modify the translation table in a desired way, several parameters must be available. First, there must be a population of bit strings together with their corresponding symbol strings and their fitness values (as shown in figure 3.7). Furthermore four more parameters especially designed to control the adaptation of the translation table must be set by the user.

1. **amount of table spaces to be updated**
   The amount of table spaces to be updated must be defined as a percentage of all the available table spaces. This parameter decides how many parts of valuable individuals are made available to the rest of the population within one generation. To support the elitist strategy used here a minimal amount of table spaces must be left unchanged. That amount must equal to the amount of table spaces that maximally can be accessed by one individual[26]. A larger amount left to be unchanged is often advantageous.

2. **preferred length of substrings**
   This parameter sets a preference on the length of the symbol substrings (modules) that are chosen for entering the translation table. In order to keep the symbol strings (individuals) from growing or shrinking too rapidly it should be set to a value that most of the time does not grow or shrink the modules captured in the translation table (the length has to be set in percentages of the total length of a symbol string individual)[27]. By setting this parameter it is also possible to influence the length of optimization on a fixed sized coding before growing or shrinking the coding[28], but this should be done by setting the parameter presented next.

3. **enforcing the preferred length**
   The purpose of introducing this parameter is to give the user the possibility to take influence on how long a fixed length coding will be

[26] If e.g. the bit string length is 30 bits and 6 bits index one table space, then at least $\frac{30}{6} = 5$ table spaces should remain unchanged.

[27] Let's assume that the symbol strings consist of 5 table entries. Then the preferred length should be set to 20% so that each time a table space is replaced it most likely will not cause the individual to grow or shrink.

[28] This can be done by setting the length to shrink while it should grow or vice versa. This adds difficulty to evolve the length of the coding to the right size, which enables the algorithm to search longer using a fixed length before extending or diminishing it.

optimized, before it will be extended or diminished. The smaller the value of this parameter the longer this local search is active or actually the higher the *probability* that the coding will not change in size. The parameter sets the variance of the Gaussian used to chose the length of the substring (module) as will be explained later.

4. **mutation rate for symbol substrings**
   Mutation on selected symbol substrings is a necessary action to keep the translation table from unrecoverable loss of information. The mutation rate should be kept low for similar reasons as it is kept low for the SGA and the ESGA.

After all those parameters are known the translation table can be modified. The modification takes three steps. First the table entries, which will be modified, must be selected. Then modules (substrings) must be chosen that will replace the old table entries. Finally the chosen substrings must be inserted in the selected table spaces.

Selection of Table Spaces

The first step[29] is to find the table spaces, which should be updated with new substrings. The method of choosing the places is driven by two ideas:

1. Keep entries that are mostly used by the best individuals unchanged as they seem to contain valuable information and as they are needed in order not to loose the solutions already found.

2. Replace entries that are only used by individuals of low quality (or are not used at all[30], which rarely happens, if the population is big enough), as the information stored in them does not seem to be relevant to the solution of the problem.

In order to satisfy those two ideas the *importance* for each table space is calculated. In order to establish the importance of each table space the population is analyzed and an *importance table* is created (see figure 3.9).

The importance of each table space is calculated by looking how often and by which individuals it is accessed. The importance table has the size of the translation table and is indexed in the same way. First the importance table is initialized with zeros. Then each time a table space (index) occurs in an individual the fitness of the individual is added to the importance value of that index. The more often an index occurs and the better the individuals,

[29] It actually can be done parallel to the second.

[30] This is a decision, which can be disputed as the elimination of information, which has not been characterized as being important or not important may actually cause loss of vital information. This lost information can be regained through mutation, but this may prolong the optimization process unnecessary.

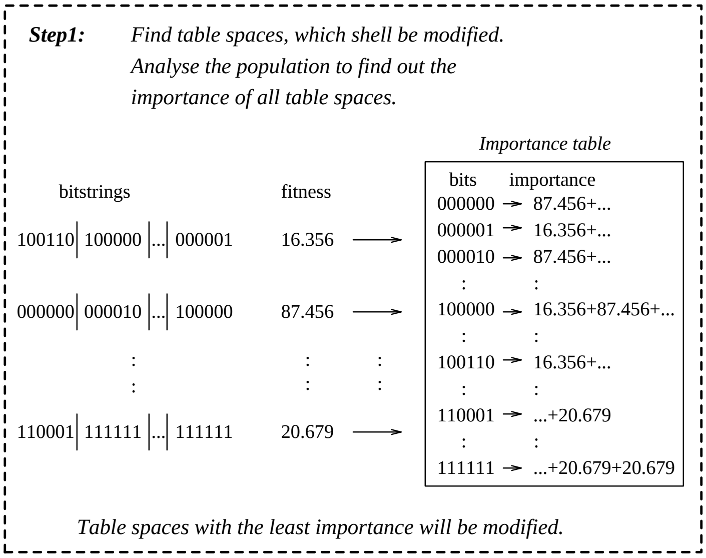

Figure 3.9: The creation of the *importance table.*

who use it are, the higher is the importance value of that index.
Finally all places with the lowest importance values are selected for modification. This method does not ensure an elitist strategy anymore[31].

Selection of Modules

The second step is to choose the substrings that will replace the selected entries of the translation table. An evolutionary approach has been chosen to accomplish this task. The main idea is to choose substrings (modules) from symbol strings (individuals) of high quality (see figure 3.10).

First there are as many symbol strings (individuals) selected to donate a substring as there are table spaces which need to be updated. In order to support the idea of highly fit individuals donating the modules, on the one hand, and in order to keep the algorithm a global search technique, on the other hand, *roulette wheel selection* is used to draw donors.

After a group of donors has been established (some donors may be identical due to roulette wheel selection), each donor contributes exactly one substring that will replace an existing table entry. At this point the length and the starting point of each substring that will be donated must be found.

The length of the substring may range from zero (an empty substring) to the length of the donor string. In the case that the length is equal to the length of the donor string, the donor string is treated as being a module. If the length would be chosen arbitrarily, this would lead to modules being

[31] The table entries accessed by the best individual might be overwritten, when e.g. the best individual is the only one accessing those entries and many individuals of just slightly lower quality exist.

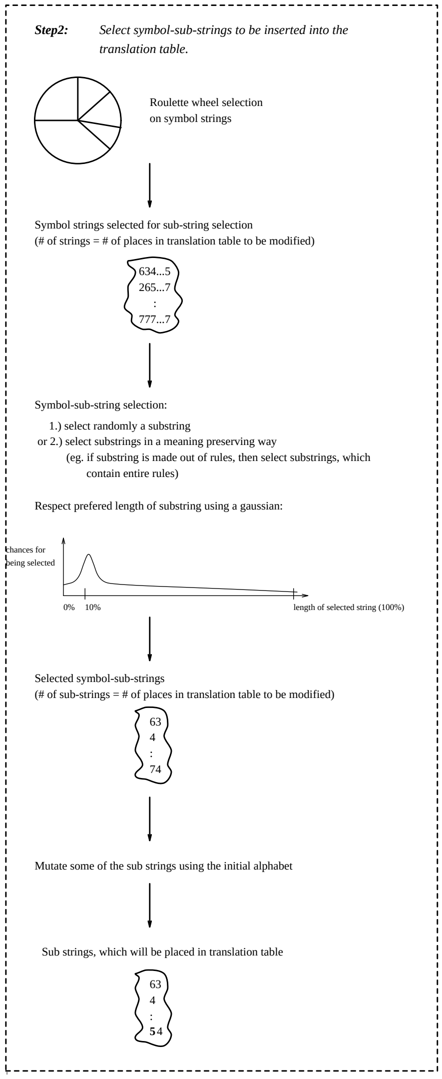

Figure 3.10: Selection of modules

on average half the size of the donor symbol string. This is usually *not* a desired feature as it would cause the genetic coding of the individuals to constantly grow[32]. In order to be able to influence the growing (or shrinking) of the individuals the length of the substrings are chosen with a Gaussian probability. The preferred length parameter defines the position of the peak of the Gaussian and the enforcement parameter defines the variance of the Gaussian (see figure 3.10). The peak decides *which* length is most probable to be chosen and the variance decides *how* probable the preferred length is to be chosen in comparison to all other possibilities.

If one does not want to force the genetic coding of the individuals to grow or shrink in length, the peak should be set in a way that the average length of the entries, which are already in the translation table, is the preferred

[32]unless the bit string addresses just a total of one or two table entries

length. Given this case, the variance controls how likely the genetic code will grow (or shrink). The smaller the variance the less likely the size of the coding will change.

The starting point of the substring is chosen arbitrarily from all possible points. Only those points are considered being possible, which allow the string to maintain its selected length treating the string as a *linear* structure with a beginning and an end[33]. This imposes that the longer the substring is the fewer points can be chosen to be starting points.

After establishing the group of symbol substrings (modules) those substrings have to be mutated with a certain probability in order to ensure that *no* important information is unrecoverably lost[34]. Mutation is done by changing single letters of the symbol string with a certain user defined probability. If a letter is chosen to be mutated, it is exchanged with a random letter of the alphabet.

At this point the group of modules (substrings) is ready for insertion into the translation table.

Insertion of Modules into the Translation Table

Once the table spaces and the modules are established the last step that remains is the insertion of the modules into the table. There are many possibilities how to insert modules into the table and the easiest was chosen here. The modules are inserted randomly. Of course, there might be more clever ways to do that by using some kind of heuristic[35], but the investigation of all the possibilities would go beyond the scope of this work.

[33] One might wonder why the strings are being treated in a linear way and not in a circular one so that the beginning of the string is *not* adjacent to its end. Especially since they were being treated as if they were circular by choosing a two point crossover and not a one point crossover on bit strings. The explanation of this peculiar choice lies in the intend to use the SCGA to construct topologies of *feedforward* neural networks. Feedforward neural networks in fact are "linear" structures, rather than circular ones, as they have a unidirectional information flow (from input layer to output layer). Thus the choice of the linear structure approach at this point seems to be straight forward. The crossover operator on bit strings was being chosen to work with two points rather than one, as the dynamics of such an operator are increased. It can simulate a one point crossover, but additionally it is also capable of taking advantage of vital parts, which are located in the middle of an individual (bit string).

[34] Loss of information could occur, if mutation was not used. This can be shown on a very simple example. Let's say that the most important letter of the alphabet is *not* being accessed by the initial random bit string population. Now it is almost sure that this letter will be eliminated from the symbol table during the first generation. Once a letter is completely removed from the symbol table it can only be reintroduced again by using mutation.

[35] E.g. it might be advantageous to replace modules using some kind of similarity measure and inserting modules into spaces which already contain similar modules.

3.7.6 Analysis of the SCGA

This section discusses the features introduced by the add–on extension of the SCGA and their influence on the optimization. First the criteria defined in section 3.6 are reflected and then some negative and positive features of the SCGA are mentioned.

Criteria analysis

At this point the SCGA is analyzed for the features demanded in section 3.6. Some of the features could be met better than others and therefore a quick description is given.

1. **controlled growth of the genetic code**
   The SCGA has additional parameters that give the user an opportunity to take influence on the speed of the growth of the genotypes during the optimization. The variance parameter is especially important. The user can set the probability with which the genotypes will grow or shrink directly.

2. **automatic module definition**
   Modules (translation table entries) are defined automatically and their quality is ensured by selecting them according to Darwinian evolution[36]. Once a module is defined, it can be referenced many times.

3. **extended module distribution**
   Automatically defined modules can be easily accessed by all individuals of the population. Matter fact, individuals of lower fitness are more likely to access newly defined modules faster, as those table spaces are modified, which are primarily used by poor performing members of a current generation. This attempts to raise the overall fitness of the entire population.

4. **module isolation**
   The SCGA approach enables module isolation, because all modules are located in a central translation table. Thus it should be possible to find vital modules by analyzing the population[37].

5. **automatically meeting the proximity criterion**
   This criterion is not met explicitly by the SCGA. However since a dynamic change of representation between the lower (bit strings) and the higher (symbol strings) coding is enabled the genetic algorithm has

[36] This does not ensure that *every* defined module is of high quality, but the probability is high that a module is vital, as most modules are parts of individuals that have a high fitness.

[37] Check which modules are used by the best performing individuals.

the possibility to adjust the coding in such a way that the proximity criterion is met[38].

6. **supporting closure**
   Closure of the genetic coding is not more supported by the SCGA than it already has been supported by the ESGA. Basically this means that the closure problem has to be solved by the higher level genetic coding[39].

7. **integrating knowledge**
   This is a feature that is widely supported by the SCGA and therefore it is quite easy to include a priori knowledge. One can define modules and include them in the initial alphabet. Additionally, by setting the occurrence percentage it is possible to decide how important and dominating a module should be. Due to mutation (of substrings selected for entering the translation table) the module stays accessible throughout the entire evolution.

Positive and negative features

The criteria described in the previous section introduce the major properties of the SCGA, but there are also additional features that are worth while mentioning.

Firstly there are two genetic operators that are used implicitly and have not been mentioned yet. The operators are *duplication* and *deletion* [9, 10]. Duplication and deletion are used implicitly as they are not directly applied to the genetic code. Yet, the process of automatically defining modules makes use of them. The way in which modules are constructed is by copying already existing information into the translation table and this is a form of duplication. On the other hand, it is also possible to insert an empty module into the table and this either immediately leads to or at least enables deletion[40].

Furthermore there is one other criterion (next to modularity) from the set of criteria described in section 3.2 that is partly favored by the SCGA. The criterion is compactness. At first sight it seems to be contradictory that the SCGA should be well posed concerning compactness, as it uses duplication as one of the main genetic operators. Nevertheless compactness

[38] It is not explicitly forced to meet the proximity criterion, but translation tables that let the lower level genetic coding meet this criterion enable a faster improvement of individuals and are therefore more likely to survive in its essential parts (parts not modified by table adaption).

[39] Actually there are several possibilities to ensure closure. One can make sure that once a crossover point is selected a point of the same type is selected in the other individual [15]. Another possibility that can be supported by the SCGA is to make sure that all the selected modules in the translation table have the same type of starting and ending points.

[40] If an empty module is chosen to enter the table and the table entry (which contained a non–empty module) is accessed by an individual then this individual experiences deletion.

is enhanced due to the central translation table. On the one hand, the table itself might contain many similar (or even equal) modules, but, on the other hand, all individuals are using the table by referencing it[41], thus using the same information in several different places without explicitly rewriting it.

Another positive feature of the SCGA is its ability to find small building blocks (modules) and use them to build larger ones in a repetitive way. This is due to the way modules are defined.

The major disadvantage of the SCGA is the fact that each generation more evaluations must be performed than with a simple genetic algorithm. On the one hand, up to two times as many individuals must be evaluated for their fitness each generation and, on the other hand, the overhead for the genetic algorithm[42] increases. The overhead however can often be disregarded, if the fitness evaluation takes much longer[43] than the overhead calculations. Nevertheless the fact that many individuals more must be evaluated remains.

Another problem that may arise is that due to the global translation table the population might converge to a local minimum and encounter great difficulties in escaping from it. Whether this problem will arise or not is still unknown at this point, but will be considered later.

3.8 Extended Semantic Changing Genetic Algorithm

The extended semantic changing genetic algorithm (ESCGA) is an extension of the semantic changing genetic algorithm (SCGA). The idea for the ESCGA emerged from the combination of the extended simple genetic algorithm (ESGA) and the semantic changing genetic algorithm (SCGA).

The modifications of the SCGA are discussed just briefly here, as the first test results have already revealed that many of the positive features of the SCGA seem to suffer from the extensions. The ESCGA seems to be a compromise between the ESGA and the SCGA and it does not quite reach neither the performance of the ESGA nor of the SCGA depending on the class of problems, as will be shown in section 3.9.

3.8.1 Extensions

There are basically just two modifications when going from the SCGA to the ESCGA.

[41]This also imposes a relationship between the size of the population and the size of the symbol table. The table size actually limits the size of the population, because if there is a very small table and a large number of individuals, the individuals are likely to be equal (or very similar).

[42]The overhead now includes an analysis of the population and the translation table adaptation.

[43]as it actually often does for many problems including neural network evaluations

The first and major difference is that the ESCGA works on bit strings of variable length (like the ESGA). Other than that its genetic loop is pretty much the same as for the SCGA.
The only other difference is that the parameter which defines the preferred length of substrings (modules) must be updated in order to prevent the genotypes from growing or shrinking exponentially. In the beginning a preferred length of substrings is defined by the user. If it is desirable (the usual case) to keep the length of the genotype most likely to stay constant one can calculate the preferred length using the following formula:

$$prefered\_length = \frac{100 \cdot bits\_per\_symbol}{length_{bitstring}}\%$$

This formula then determines the percentage (of the length) of the symbol string which will be preferred when choosing substrings[44] (modules). With bit strings of different length however this must not stay constant. Otherwise, once a bit string grows or shrinks it will become very likely that it will continue to do so. In order to prevent this from happening an invariant is calculated using the initial bit string population an the initial preferred length parameter[45]. This invariant[46] represents a growing or shrinking probability which is then imposed on every module selected. It stays constant throughout the entire evolution and allows to dynamically establish the prefered length of modules for individuals of different bit string sizes.

3.8.2 Problems

The described extensions cause (at least) a couple of problems that are briefly mentioned here:

- **lost growth control**
  The control of the growth of the genotypes is lost when using variable length bit strings, as there is no specific parameter that can be set to control the growth of the bit string coding (and therefore also the growth of the symbol strings).

[44] Let's assume we have a bit string that is 30 bits long and six bits define a translation table entry (bits per symbol) then the formula would state that 20% of the resulting symbol string are most likely to be encapsulated in a module (table entry) during table adaptation. If all table entries have the same length, then the genotype is least likely to be modified. If the table entries differ in length then the average length of all entries defines the most likely length of the module.

[45] All individuals in the initial population must have the same length.

[46] If the invariant is 1.0, no growing or shrinking is preferred (the usual case). If it is smaller than 1.0, than the genotypes are likely to shrink and if it is greater than 1.0, they are likely to grow. It is calculated by the following formula:

$$invariant = \frac{initial\_prefered\_length \cdot length_{bitstring}}{bits\_per\_symbol}.$$

- **interfering operations**
  Additionally two operations seem to contradict each other now. Whenever the genetic algorithm tries to grow or shrink the length of a genostring it is being disturbed by the adapting growth probability (preferred module length) that tries to compensate the change.

Those problems and the initial tests (as we will see) suggest that this algorithm is a compromise without great advantages.

3.9 Tests

This section presents the initial tests that were performed in order to check the general abilities and specific advantages of the newly introduced semantic changing genetic algorithm (SCGA) and the extended semantic changing genetic algorithm (ESCGA) in comparison to the simple extended genetic algorithm (ESGA). Three different tests were conducted and interesting results were obtained. The tests represent different problem classes. The first two present an unimodal and a multimodal function. The third test has been conducted to show the local optimization capability of the SCGA. The tests were furthermore designed to investigate the growing behavior of the genotypes for the three different approaches. During these tests the genotypes had to grow *at least* 50 times to reach the optimal solution. This can be considered to be a very extreme demand, as for most problems it should be possible to estimate the dimension of the problem much better so that such a growth would not be necessary. Nevertheless this was done intentionally in order to clearly elaborate the different behaviors. Furthermore all test had been repeated a fair amount of times so that a statistically save evaluation was possible[47].

3.9.1 Quality measures

It is necessary to explain the chosen quality measures and their validity, as some of them are not obvious. The measures are designed to reflect the speed of convergence as well as the quality of the solutions found by the different approaches. Five different measures were used:

1. **average convergence speed**
   This measure displays the amount of evaluations of individuals needed on average to reach a solution.
   However, measuring the speed of convergence by merely looking at the amount of individuals that have to be evaluated is only a valid measure, if the time needed to calculate the fitness for an individual

[47]Due to the random initialization some noise is introduced to the results of a single optimization run.

greatly exceeds the genetic overhead introduced by the SCGA. This indeed is expected to be the case when evaluating neural networks[48] and thus the measure is meaningful.
In the case that no perfect solution was reached within a defined time (number of evaluations) the measure shows during which evaluation the current best solution was found. This is done to be able to relate the following measures more accurately.
Furthermore the measure has just a limited accuracy, as the entire population was evaluated each generation[49].

2. **median convergence speed**
   The median convergence speed shows the expected amount of evaluations of individuals one has to go through in order to reach a solution. Together with the average convergence speed it is possible to estimate how often an optimization attempt will quickly lead to a solution. There are three cases:

   (a) If the median is much smaller than the average then in most cases a solution will be found quickly, while sometimes it will take very long.
   (b) If the median is much larger then the average, it is likely that only sometimes a solution is found quickly and most of the times it takes longer.
   (c) Given both values are nearly the same, then a solution is reached in the displayed amount of evaluations almost all of the time or at least there is an equal chance that it will be reached faster or slower.

3. **average quality**
   It is the quality of the best individual that is reached on average after evaluating the amount of individuals displayed by the *average convergence speed* measure.

4. **median quality**
   This is the median quality that is reached after evaluating the amount of individuals displayed by the *median convergence speed* measure.
   Again this value reflects the quality that can be expected to be reached after a certain amount[50] of evaluations (of individuals). The same kind of cases as explained for the *median convergence speed* apply here, if one wants to analyze the combination of the two quality measures.

[48] It is expected to be that way, as the algorithm presented in this research uses a learn algorithm to train the networks in order to find their fitness. If complete networks (with all the weights) were created and then just tested for their fitness, then the genetic overhead might had to be incorporated into the measure. The same is also true for the simple examples presented in this section.

[49] The size of the population sets the accuracy. The greater the population the least accurate the measure. Nevertheless this can be ignored in most cases at it lies within the noise range of the random initialization.

[50] displayed here by the median convergence speed measure

5. **percentage of perfect solutions**
This measure shows the amount of perfect solutions found within a maximal given amount of evaluations (of individuals).
This is perhaps the weakest measure as it is directly influenced by the a priori set maximal amount of evaluations (of individuals) permitted. Nevertheless it helps comparing the algorithms as it is a very practical measure[51].

3.9.2 Settings

This section describes the alphabet and all the parameters that needed to be defined a priori to each evolutionary run and that were kept constant for all the tests here.

The alphabet was set to contain the integers from zero to seven [0..7]. All integers were distributed equally (each eight times) in the initial translation table. When the ESGA was used the eight numbers were encoded binary[52] and three bits at a time were decoded into one integer (symbol).

The parameters needed by all algorithms were set to:

Population Size	20
Initial Bitstring Length	30
Bitstring Crossover Rate	0.8
Bitstring Mutation Rate	0.01
Generation Gap	1.0

The parameters additionally used by the SCGA and the ESCGA were set to:

Bits addressing one table entry[53]	6
Table Adaptation Rate[54]	0.8
Table Adaptation Mutation[55]	0.01
Preferred Length of Modules	0.2
Gaussian Variance for enforcing the Preferred Length	0.01

All those parameters were set equally for all tests unless indicated otherwise.

[51] It is practical as an experiment is usually run for a fixed amount of evaluations.

[52] One could expect to improve the performance of the ESGA by gray–coding the integers. This was avoided in order to be more realistic, as an optimal coding is often unknown.

[53] Defines implicitly the size of the translation table to be $2^6 = 64$.

[54] Defines how many percent of total table spaces are updated each generation.

[55] Sets the probability for a selected module to be modified before it is inserted into the translation table. The probability is per symbol so that a single module might be modified several times if it consists of many symbols. The module is modified by replacing one of its symbols with an entry from the alphabet.

Furthermore the fitness is defined to be positive and best when zero. The larger the fitness the worse the performance of the individual.

Additionally *all* test results for the speed of convergence might be diminished by 20% as the crossover rate was at 80% and thus on average 20% of the individuals of each generation did not have to be reevaluated[56]. However since *all* the test results suffer from this incorrectness, it should not matter much when comparing them with each other.

A different amount of tests was conducted for the different algorithms. For the ESGA 50 tests (for each problem) were considered being sufficient. For the SCGA and the ESCGA about 2,500 tests each were made (for each problem). This is due to the fact that The SCGA and the ESCGA had to be tested additionally for different initializations of the translation table next to different initializations of the random initial population. 50 different random table and population initializations each were created so that for each table initialization 50 different initial populations were tested[57]. For the ESGA only 50 different initial populations were regarded.

3.9.3 Test 1

The first test was designed to show the performance of the SCGA on a problem where many of the special features of the SCGA[58] should not be exploited. The test was designed to show whether the SCGA is capable of solving basic genetic problems at all. All the algorithms were allowed to run for 8,000 fitness evaluations, which in this case did not really matter as most evolutions found the optimal solution much earlier.

The function to be optimized was simple:

$$f(x) = (total\_of\_digits - 250 \cdot 7)^2$$

The optimum is a number with a total of digits equal to $250 \cdot 7$. With the initial population containing just individuals with a symbol string length of 5 and the maximum value of a symbol being 7 the symbol strings had to grow at least 50 times to find an optimal solution.

Table 3.1 gives a first impression of the performance of the three different approaches. It shows how often the problem was solved by the competing algorithms within the permitted time. The results demonstrate that all of the candidates solved that problem almost always. Actually the ESGA performed slightly better according to this measure.

[56]All individuals were evaluated at the beginning of a new generation.

[57]Less than 2,500 test runs were taken into consideration during analysis as some test runs were not valid due to hardware problems (e.g. someone booting the computer in the middle of a run).

[58]The most important feature that was not contributing anything advantageous was the automatic module definition. There was no pressure from the fitness function that would force the modules to contain just a special part or/and arrangement of information.

algorithm	percentage
ESGA	**100.0%** of 50 tests
SCGA	99.83% of 2200 tests
ESCGA	97.96% of 2450 tests

Table 3.1: Percentage of successful optimizations for the first problem.

algorithm	median time	average time
ESGA	**1,000**	**1,026**
SCGA	2,599	2,815
ESCGA	2,760	3,002

Table 3.2: Median and average amount of fitness evaluations needed to solve the first problem.

Nevertheless there are some differences in the time the algorithms needed to solve the problem as is shown in table 3.2. The table demonstrates that the ESGA found a solution much faster than any of its competitors. The SCGA needed approximately 2.5 times longer and the ESCGA was even slower than the SCGA in finding a solution.

Table 3.3 finally shows the actual qualities of the solutions and supports what has partly been indicated by the percentage measure. Again the ESGA yields a slight better performance than the other approaches, but all optimization runs end in either optimal solutions (fitness of 0.0) or in solutions very close to the optimum.

Analysis

This first experiment shows that the ESGA is more powerful in solving this kind of problems than the newly introduced approaches as it solves them much faster (by more than a factor of two) and with a higher accuracy (solutions were always optimal). Nevertheless the SCGA and the ESCGA are both capable of solving the problem, even if they are slower. The aim was

algorithm	median quality	average quality
ESGA	**0.000000**	**0.000000**
SCGA	0.000000	0.001739
ESCGA	0.000000	0.020408

Table 3.3: Median and average quality of the solutions.

not to show that the SCGA and ESCGA are more powerful in solving this problem, but that they are at all capable of solving it. It is not surprising that both performed worse, as their special features were not needed to solve the problem, as it was too easy. Thus the overhead of evaluating about twice as many individuals per generation compared to the ESGA forced the SCGA and ESCGA to consume more time.

3.9.4 Test 2

The second test was constructed to demonstrate the power of the extensions introduced with the SCGA. Again the genotypes had to grow 50 times, but this time the function to be optimized was designed in a way that building blocks (vital modules) could be found at a level, where the length of coding has not grown yet and at any other point in the evolution of the genotypes. Additionally those building blocks could be used to build larger modules and those larger modules could then be used as building blocks for even larger modules etc. In fact there was more fitness pressure applied to finding a local optimum then to growing the genotype[59]. Nevertheless once a local optimum was reached, the genotype had to grow in order to leave it. Finally the function was designed to have a neutral element that did not have any direct influence on the fitness[60]. A function which satisfied the described requirements was chosen to be:

$$fitness = (250 \cdot 7 - total\_of\_digits)^2 + (250 - amount\_of\_7s)^4$$

The best possible fitness is 0.0 and is obtained by individuals that contain exactly 250 times the digit 7 and an arbitrary amount of zeros. The worst fitness is:

$$worst\_fitness = 3.062500 \cdot 10^6 + 3.906250 \cdot 10^9 = 3.909312 \cdot 10^9$$

It can be seen that there is more pressure on finding the right amount of 7ths then there is on just growing the genotype arbitrarily. Of course, the fitness will improve with the growth of the string (at least up to a certain point), but it will improve much more rapidly, if the growing genotype contains modules of the form $777\ldots7$.

Several tests were done to elaborate the difference between the performances of the ESGA, the SCGA and the ESCGA. The ESGA was tested twice, the first time the time limit was set to 8,000 fitness evaluations and then, due to the poor performance of the algorithm, it was increased to 80,000 fitness evaluations. The SCGA was tested under the usual conditions[61], as there

[59] In other experiments that delivered similar results the pressure was distributed equally.

[60] This was done to check the possible influence of such elements.

[61] The time limit was set to 8,000 fitness evaluations and all the parameters were set as in the previous test and as described in section 3.9.2.

algorithm	percentage
ESGA 8,000	0.00% of 50 tests
ESGA 80,000	0.00% of 20 tests
SCGA	**13.0%** of 2200 tests
ESCGA	0.00% of 2450 tests
ESCGA adjusted[62]	2.04% of 2050 tests

Table 3.4: Percentage of successful optimizations for the second problem.

was no need for a change. The ESCGA however needed to undergo two different tests as the parameters for the preferred length of modules and the amount of table spaces to be updated had to be adjusted in order to improve its performance.
Basically, all algorithms were run under standard conditions first and then the conditions were adjusted for the poorly performing approaches in order to correctly display their ability to solve this problem.

Table 3.4 gives a first impression of the performances by presenting how often a perfect solution has been reached. The winner of this competition was the SCGA. Although it found a solution in only 13% of all evolutionary runs it outperformed greatly all its competitors. While the ESCGA was still able to find a few perfect solutions in a reasonable amount of time the ESGA was not. Matter fact, even running 10 times as long as the SCGA the ESGA was not able to come up with even one perfect solution.

In this case where hardly ever a perfect solution was found there is not much use to look at how fast it was (not) found and therefore the table with the convergence speed measures is skipped for this test.

The quality that was reached by the best individuals however is of great interest, as it shows how well the approaches really performed.

Table 3.5 lists the best median and average qualities reached by each algorithm. It can be clearly seen that the performance of the SCGA is outstanding compared to the performance of all other approaches. The quality of the solution reached by the SCGA is in at least 50% of all runs excellent (2.0 or better) and magnitudes better then the best next performance given by the ESCGA. The ESGA is obviously extremely slow in solving this kind of problems, as even running 10 times longer it solved the problem far worse than the SCGA. Nevertheless the solution found by the ESGA is much better than at the beginning of the optimization (where it was approximately $3.9 \cdot 10^9$).

[62] The preferred length of modules was set to 8%. This means that the genostrings were pushed to shrink, while the fitness function pressured them to grow. Pushing the strings to shrink gave them an (poor) opportunity to optimize the contents before growing.

algorithm	median quality	average quality
ESGA 8,000	551,345.0	721,004.0
ESGA 80,000	56,080.0	92,254.4
SCGA	**2.0**	**691.6**
ESCGA	19,562.0	3,940,344.4
ESCGA adjusted	2,834.0	215,349.6

Table 3.5: Median and average quality of the solutions.

Analysis

This test demonstrates that, if the newly introduced features of the SCGA are useful in optimizing a function, there is a great speed up in finding a sub optimal solution. The extensions of the ESCGA however do not seem to be advantageous[63]. Although the ESCGA passed the test better than the ESGA, it seemed to be impossible to adjust the parameters in a way that it would be able to compete with the SCGA. Furthermore the adjustment of the parameters is not a simple task and therefore the transparency[64] of the algorithm is limited.

In order to investigate the results more thoroughly one can compare the typical best individuals of all tests. The typical best individual that was produced by the SCGA after 8,000 fitness evaluations looks like this:

777777777777777777777777277777777777777777770777777777777777777777
777777777727777777777777777777777777777777777777777777777777777777
777777777777777777777777777777777777777747777777777777777777777777
7777777777777777777777777777777777777777777777777777777

It represents a solution that is very close to the global optimum (sub optimal). The genotype contains 249 times the digit 7, its total of digits is 1751 and its quality 2.0. Curiously the string contains only one neutral element (0). This is due to the fact, that although there was no penalty nor credit given for zeros, the fitness function pushed the algorithm to collect 7th. It was accomplished by the controlled growth of the genotypes. The slower the genotypes were grown the more likely they contained only the correct information (digit 7), as they had more time to find the local optimum which they were limited to by their length.
The ESGA, on the other hand, did not benefit from a controlled growth and therefore found the following typical individuals after 8,000 fitness evaluations:

775777733765770737357777707710377043772300773731377237531735017760

[63] This is at least partly due to the problems mentioned in section 3.8.2.

[64] Transparency is a criterion or measure for how easy to use the parameters of a certain algorithm are designed. If it is simple to find good values for whatever problem is to be solved, then the algorithm is transparent.

7714543625777710336264777777777177616447777775617772377614752131703
3775617772377614756171703372763557507472765777377571775017770771 41
5041776257777143777657733375717350177617771455763777770731572504176
1077714776375677777717671445777773171777237761477417170337277757 77
0747376577737757177501777176141572257777100376247752412755437652 70
7277757727377757077503771477723074635257373375317750177607716177 22
47777101772247710543775027212707775777707177342777122726710 7141

This example indicates the problem that arises when no growth control is enabled. Although the individual contains almost all (236) the 7th required it also carries many other digits (fitness = 572,777.0). All other digits contributed to solving one part of the fitness function at the beginning of the optimization and were regarded to have a positive influence, but at the stage represented by the individual displayed the algorithm must now get rid of them. This is not easy, because those digits are spread throughout the entire string so that the task becomes hard for the crossover operator. Nevertheless the ESGA is capable of finding the global optimum, it just takes very long. An individual after 80,000 string evaluations looks typically like that:

720777721077174773777757000777707727777147071777775003770074277147
717777727777707177777302072277727711707732707777377777020777771310
477037707777700157777727773717177751167771777710777170777772167737
774377777277677377777417177772277727711707532304777337773000777770 2
304772777077777001737777277777171777511777717777107771707777721377
37774177737277777777777172777773777377734772730 7070

The amount of 7th has not changed[65], but there are now less other digits in the string. The individual has a fitness of 62,132.0.

The performance of the ESCGA lies between the performance of the SCGA and the ESGA. An individual found after 8,000 evaluations without any adjustments of the control parameters looked like this[66]:

73774777777777777704377077727770431777777777707772777043177777777777
77377477777777777770437707772777043177777777777777377477777777777704
37707772777043177777777777777377477777777777770437277707772777043177
77777777777377477777777777777043770777277704317777777777777737747 7777
777777704372777761537777577777747777777777773777757 77777

It contains 243 times the number 7, but nevertheless its quality is only 19,562.0. After adjusting the parameters in order to enforce a slower growth of the genotypes the quality of the solutions improved. An example for such a solution is:

[65] This is true for this example and it is true for most examples that the amount of seventh will not change drastically. This is due to the fact that, on the one hand, the total of digits exceeded the desired value and therefore more 7th are unlikely to be created and, on the other hand, 7th should not disappear as this would have a bad influence on the second part of the fitness function.

[66] In this experiment the best individuals looked quite different from each other and this one is just an example of how a genotype looked.

1070057077007701707577770770770707077770770770707077770570770707700570770075707700777077770707707707707707707707707707070770777077077075070077077070707777057077070770770770770707077770770770707077707770770077707777077077070707707707770700777050707077563777077077070707777077077070707757077007770770770770770707077005707700770177707707707070777570707777070777707707707075707770770770707077770570770
77077

This individual contains 245 times the digit 7 and its overall quality is 2,834. Curiously this individual contains the neutral element zero quite often, a phenomenon that occurred sometimes with the ESCGA, but not with the SCGA. The explanation lies again in the controlled growth. The growth could not be controlled as strictly for the ESCGA as it could be for the SCGA. An indication that supports this thesis is also the fact that it seemed impossible to adjust the control parameters in such a way that the ESCGA was competitive to the SCGA.

3.9.5 Test 3

This test was done out of curiosity. The question was, what would happen if just the second part of the previous fitness function was used. The goal was to show what will happen, if instead of having neutral (zeros), positive (seventh) and negative (all other digits) information there would be only neutral and positive information available. Thus more global optima were created that the algorithms could reach. This, of course, made the problem easier, but now the interesting part was to see which global optima would be reached by the different approaches.

The fitness function was the following:

$$fitness = (250 - amount\_of\_7s)^2$$

The best fitness was again 0.0 and reached by all individuals that contained the digit 7 exactly 250 times.

The results were surprising. All the algorithms always found perfect solutions, but although the ESGA outperformed the SCGA and the ESCGA in the speed of convergence, as shown in table 3.6, the SCGA found the most compact genotypes for the optimal solutions[67]. A comparison of the final solutions illustrates that.

Typical best individual found by the SCGA:
771776777777777777777777777777777777777777777777777777777777
777777777776777777177777777777777777777777777777177777777777
777777777777777777777777777677777777777777777777777777777777
7777767777777777777777777777777777777277777777777776777777

[67]The genotypes were always of sub optimal compactness.

algorithm	median time	average time
ESGA	**1,160**	**1,233**
SCGA	2,093	2,111
ESCGA	1,480	1,684

Table 3.6: Median and average amount of fitness evaluations needed to solve the third problem.

Typical best individual found by the ESCGA:
27742242747727272772227777727772227447447772247774126477772727773
22744727447772227277422427477272727722277777277227742242747727272
772227777772777222744747772777274477722477741264777227727772272727
72777224777427777777727772772427477272712772274477722477722277222742
222774224274772727277222777772772277777772737774272227272727227747
772274477272274477722477741264777727772227442727277732274472744272
72647777272777372227722274222277422427477272727722277777277127222
7447274427777172727122472277777

Typical best individual found by the ESGA:
353627561756146567342617050614754774261657774247166717417072777773
577011731747637747177377775634763774657747566735762717225617167544
277677546470236377613062637652247173577147177677633654706341705347
177356774267702470436737613471637624706767777577115771747637747177
357775634763774757747566724762707235767377161475736427546375677542
470636377653062637672247173577147176677733654706345705347177356774
267742470436737613071637424706767176130756020661734061263474775770
557707161025461734757347305163635677426655737477531717657763677713
621737567763431736750563636755147343775770576613072777613347573557
041636356774267742470476737403072777612247173577147172777637654707
241705343177643563477375717726373567746173677056363612716376517347
17677763565474735770534737

Compactness is not explicitly demanded by the fitness function and thus the ESGA finds solutions which contain many neutral elements that enlarge the genotypes greatly. The SCGA, on the other hand, finds an almost compact coding. When comparing the performance of the three algorithms the suspicion arises that, if the solution is approached slowly, it results in a more compact coding than if it is reached faster.

3.10 Conclusion

In this chapter several genetic approaches were covered. After a short introduction, the importance of the genetic coding was discussed and the following criteria were established to measure the quality of a genetic coding:

- completeness
- closure
- proximity
- short schemata
- compactness
- non–isomorphism
- modularity

Then an outline of the simple genetic algorithm (SGA) as introduced by Holland [9, 10] was presented, the function of the genetic operators – selection, crossover, mutation and reduction – that are used in this research was described and the extended simple genetic algorithm (ESGA) was derived that could operate on genotypes of variable length.

Next, the need for an extension to the genetic algorithm was elaborated. Several features were presented that are desirable, but not covered by neither the SGA nor the ESGA. The features are:

- controlled growth of the genetic code
- automatic module definition
- extended module distribution
- module isolation
- automatically meeting the proximity criterion
- supporting closure
- integrating knowledge

Then the semantic changing genetic algorithm (SCGA) was introduced. It included most of those features. The SCGA introduces an add–on extension to the simple genetic algorithm, which dynamically modifies the coding of the genotypes using population analysis and Darwinian evolution. The next step was then to introduce the extended semantic changing algorithm (ESCGA) that used the extended simple genetic algorithm as its basis. The ESCGA was initially analyzed critically and the weak points of the extension – lost growth control and interfering operations – were pointed out.

Finally, three tests were conducted to compare the capabilities of the three approaches – ESGA, SCGA and ESCGA. The tests showed that the newly introduced algorithms are capable of solving genetic problems. Furthermore they showed that there is a class of functions (represented by the fitness function in test 2) where the SCGA clearly outperforms the ESGA. The

ESCGA, on the other hand, seems to be a poor compromise between the ESGA and the SCGA. It lacks the dynamics of the ESGA and it lacks the control possibilities the user has using the SCGA.

The tests indicate that the SCGA might be more useful for solving some problems than the ESGA and that this strongly depends on the modularity of the problem, as well as on the possibility to use simple modules to construct larger ones consecutively (e.g. the way fractals are designed). In the context of neural networks this seems to be the case [11], but, of course, it depends very much on the genetic coding how well the special features of the SCGA are exploited.

Chapter 4

Evolving Topologies

In this chapter the evolution of topologies of feedforward neural networks is discussed and a promising approach to this undertaking is presented. A short overview over different strategies of designing a genetic coding of neural networks is given first. Then a higher level genetic coding is developed that is supposed to support the special features of the Semantic Changing Genetic Algorithm (SCGA) and the development of modular feedforward networks. This genetic coding provides a mechanism for encoding just the topology of feedforward networks and not the complete networks. A learn algorithm is chosen to measure the performance of the networks. Finally, the importance of an adequate fitness function is emphasized and several fitness functions are suggested and analyzed.

4.1 Genetic Coding

The importance of the genetic coding has been already described in the previous chapter. In this section categories of different genetic codings of artificial neural networks are presented and then the genetic coding used in this research is evolved. Finally, the advantages and problems of this coding are discussed.

4.1.1 Categories

Genetic codings of neural networks can be classified into different categories. Only a very abstract classification will be presented here in order to show the two major different approaches to network coding. This classification has been suggested by Yao [34, 31] and classifies genetic codings of neural networks into two categories – the *direct* and the *indirect* encoding scheme.

Direct Encoding Scheme

The main idea of the direct encoding scheme (DES) is that the phenotype (neural network) is completely defined by the genotype. This means that every unit and every connection (and its strength/weight) has its representation in the genotype. A good example of a DES coding is a coding that uses the connection weight matrix as the genotype.
The advantage of this coding is that it is easy to evaluate the performance of the networks (phenotypes), as it can be directly measured. Nevertheless the DES has also some major disadvantages. The first disadvantage is the fact that with a growing size of phenotypes the genotypes are growing exponentially, because for each additional unit of a network all its connection have to be represented in the genotype. This limits the use of the DES to small networks as large genotypes limit the performance of the genetic operators[1]. The second disadvantage is the functional block problem (FBP). As described in section 3.2, it is important to find functional blocks and code them close to each other on the genotype. This however remains an unresolved problem when using the DES[2].

Indirect Encoding Scheme

The indirect encoding scheme (IES) is more compact than the DES and closer to the biological model. The idea is to encode just the most important parts of an artificial neural network in the genotype and then use predefined rules to extract the phenotype. The IES was split by Yao into three different classes:

1. Blueprints as used by Harp et al. [4, 5, 6] that encode parameters for constructing the connectivity and important parameters for the evaluation algorithm. The connectivity parameters guide predefined developmental rules. Furthermore Harp et al. use backpropagation as the evaluation algorithm and encode the learn rate and the momentum on the genotype.

2. Coding of developmental rules as done by Mjolsness et al. [19] and Kitano [12]. The developmental rules themselves are encoded in the genotype. This is done by encoding recursive equations (Mjolsness et al.) or generation rules (Kitano).

[1]Mutation e.g. becomes quickly meaningless, unless it is permanently adjusted. To add, remove or change a single connection in a network with many thousands connections has usually barely any effect on its performance.

[2]There is e.g. a problem with isomorphism. Units on one layer act together in solving a task, but in two equal networks different units in a layer may solve different parts of the task. In a direct coding they are treated individually and thus the crossover operator may easily destroy the functionality of a layer by tearing the units apart and putting them back together. The offspring networks could still have the same amount of units in the layer and the same connectivity, but the mix of biases and weights would not implement the same functionality.

3. Fractal coding of the connectivity as done by Merrill and Port [20].

The IES is the more advanced encoding strategy. It has the advantages of a relatively small FBP and an easy to control size of genotypes. Furthermore the IES allows to make design decisions, as one has to define the "important parts of an artificial neural network" that will be encoded onto the genotype.

4.1.2 Introduction to a Genetic Coding for Neural Networks

This section presents the idea behind a new higher level genetic coding for topology evolution with the SCGA. The coding is an IES coding, but it does not fit directly into one of the sub–categories mentioned by Yao. The idea is to encode networks with a program like structure that defines a growing process of the network starting from an empty network structure[3].

The first step is to decide what the "important parts" of a feedforward network are that should be coded onto the genotype. Starting out, two parts can be distinguished that together define the topology of a feedforward network[4]:

1. the amount of input, hidden and output units
2. the connectivity of those units

In order to prevent the genotypes from growing exponentially and for some additional reasons explained later it seems more appealing to focus on encoding just the units on the genotype and developing most of the connectivity as a part of the translation process from genotype to phenotype. A decision was made to encode the units and their relative position to each other on the genotype. Thus the alphabet of the higher level genetic coding contains a symbol for a unit and symbols to define the position of the unit inside the network. Since a feedforward network is a multi–dimensional structure and the genotype is a one–dimensional structure a design decision had to be made which units were to be coded close together on the genotype. The goal was to assure that functional modular structures could be created. This led to the unit–cluster model of the network.

4.1.3 The Unit–Cluster Model

The unit–cluster model defines clusters of units where each unit in a cluster is treated equally. Units belonging to one cluster are coded closest to each other on the genotype. They all have fan-ins and fan-outs connecting them to exactly the same units and that makes them functionally equal[5]. The

[3] Another approach that is based on the same idea can be found in [11].

[4] More parts can be found such as e.g. the activation function(s) of the units.

[5] All units in such a cluster can implement equally well all parts of the task that is being solved by that cluster.

primary ("backbone") connections between clusters are established during the decoding process of the genotypes.

This leads to a very simple system with a level register and an alphabet that consists of three letters. The level register (a positive Integer) defines the level on which a unit–cluster will be created and is initialized with 0 for the input layer[6]. The alphabet is:

$$\{\mathbf{U}, >, <\}$$

The three letters are interpreted in the following way:

- **U**
  create a unit
- $>$
  increment the level register; before incrementing: automatically define all units created since the last letter not equal to "U" to constitute a cluster and connect this new cluster to the cluster created last that is at least one level below the actual level; if there is no such cluster (e.g. level register = 0) then these units will be later defined to be input units.
- $<$
  decrement the level register, if the register value is greater than 0; before decrementing: proceed the same way as during increment
  if the level register value is equal to 0, then do not change the level register value, but define a cluster in the same way as during increment.

Examples of the Translation Process

In order to visualize the process of decoding a genotype that is made out of this alphabet a simple example is provided in figure 4.1. The figure presents a genotype of a 2–4–2 encoder and is translated in the following way:

1. the level register is set to 0; UU $\rightarrow$ create two units
2. $> \rightarrow$ increment the level register by one and define the two previously created units[7] to be a cluster. Since there is no previous cluster no connections can be made and all units in this cluster will be defined to be input units at the end of the decoding process.
3. the level register is set to 1; UUUU $\rightarrow$ create four units
4. $> \rightarrow$ increment the level register by one and define the four previously created units to be a cluster. Connect this cluster completely to the last cluster created earlier that is on a lower level

[6] It was also possible to define the output layer as level 0.

[7] all "U's" back to a letter different from U

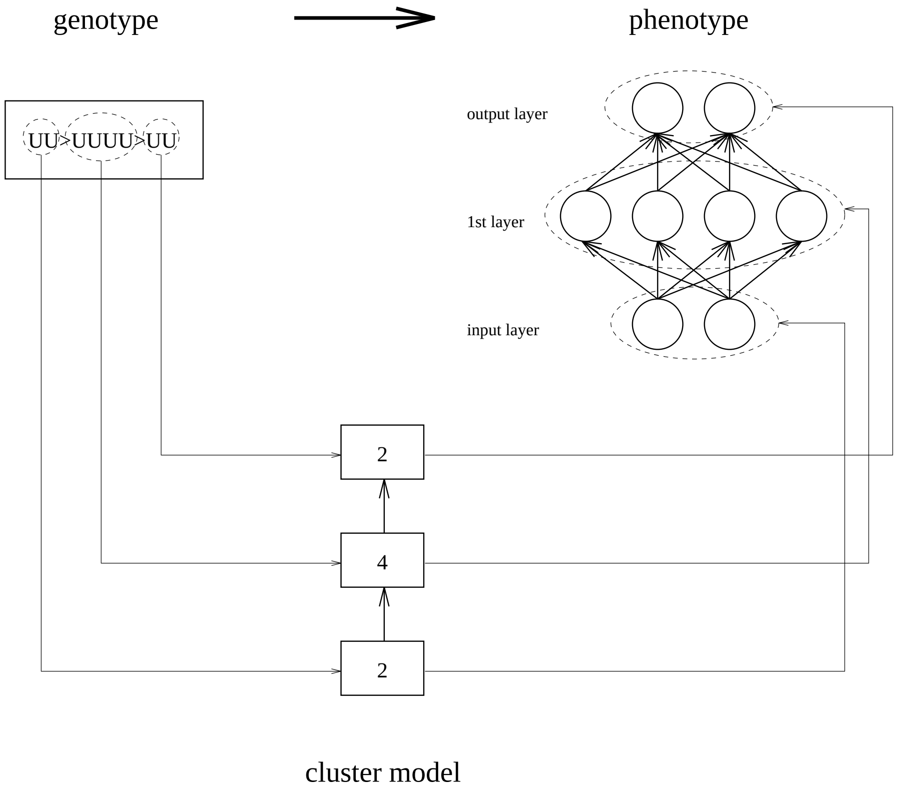

Figure 4.1: The representation of a simple 2–4–2 network using the alphabet {U, >, <}.

5. the level register is set to 2; UU → create two units
6. the end of the string is reached;
   - Define the units created last to be a cluster and connect this cluster completely to the last cluster created earlier that is on a lower level.
   - Define all units with no fan-in to be input units.
   - Define all units with no fan out to be output units
   - Assign each cluster to the right layer. A layer is defined by the *longest* way[8] between the input layer and the actual cluster.

Figure 4.2 shows those steps graphically.

A more complex genotype and its translation using the unit–cluster model is presented in figure 4.3. The genotype is translated into the unit–cluster model first. The small digits in the lower right corner of each box in the

[8]The longest way is the way that leads back from the actual cluster to the input layer though the greatest amount of units/clusters. No recursive loops are allowed in a feedforward network and thus there is always a longest way.

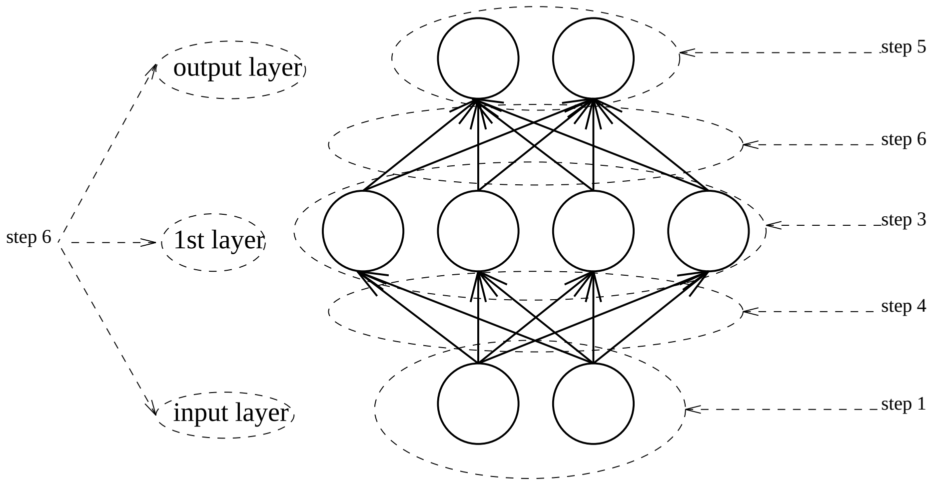

Figure 4.2: The translation process from genotype to phenotype.

unit–cluster model show the order of creation. The box with the lowest index is created first and the one with the highest index last. The unit–cluster model can be directly translated into an intermediate phenotype by expanding the clusters to units and connecting them completely. Finally, the units are sorted by layers rather then by levels. All units with no fan-ins are defined to be input units and all units with no fan-outs are defined to be output units[9]. All hidden units are assigned to the lowest possible layer.

4.1.4 Advantages and Disadvantages of the Unit–Cluster Model

In order to check, whether the unit–cluster model is useful, one must examine first, whether this genetic coding meets the criteria established in section 3.2:

1. **completeness**
   This first and very important criterion is obviously not met, as no cluster can have inputs from more than one other cluster. It will however be shown later that the criterion can be met by introducing a rather simple extension to the alphabet, which does not disrupt the main idea of the unit–cluster model.

2. **closure**
   The genetic coding (higher and lower) is closed[10], because *all* genotypes (all possible combinations of the letters) can be decoded into valid phenotypes (feedforward networks).

3. **proximity**
   The proximity criterion is widely met by the higher level coding as sim-

[9] It is possible for a unit to be both, input and output unit at the same time.
[10] It is closed with respect to the definition of closure established in section 3.2.

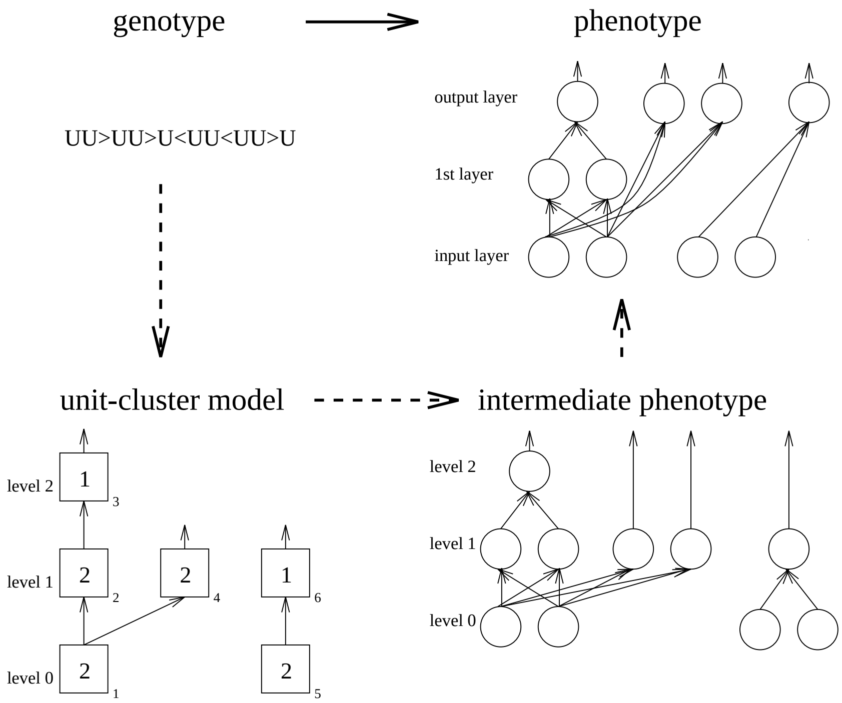

Figure 4.3: Example of a more complex genotype and its translation to a phenotype via unit–cluster model.

ilar genotypes are usually decoded to similar topologies[11]. The proximity of the lower level coding should be automatically adjusted by the SCGA.

4. **short schemata**
The highly important short schemata criterion is explicitly met by this coding. An example is shown in figure 4.4. Both, the horizontal structures as well as the vertical structures of a feedforward network are encoded on the genotype close to each other, allowing clusters, modules and sub–networks to be found as partial solutions. The priority is set on the horizontal structures and thus all units of one cluster are coded closest to each other on the genotype. Then the vertical structures are coded next closest to each other. This is done by placing the predecessor (connected to the input of the cluster) and the successor (connected to the output of the cluster) of each cluster directly next to the cluster on the genotype[12].

[11]This is disputable, because the definition of "similar" topologies is vague.

[12]If however there is more than one successor to a cluster this principle cannot be satisfied as the one dimensionality of the genotype allows only one predecessor and one successor ("backbone structure") to be coded closest to the cluster.

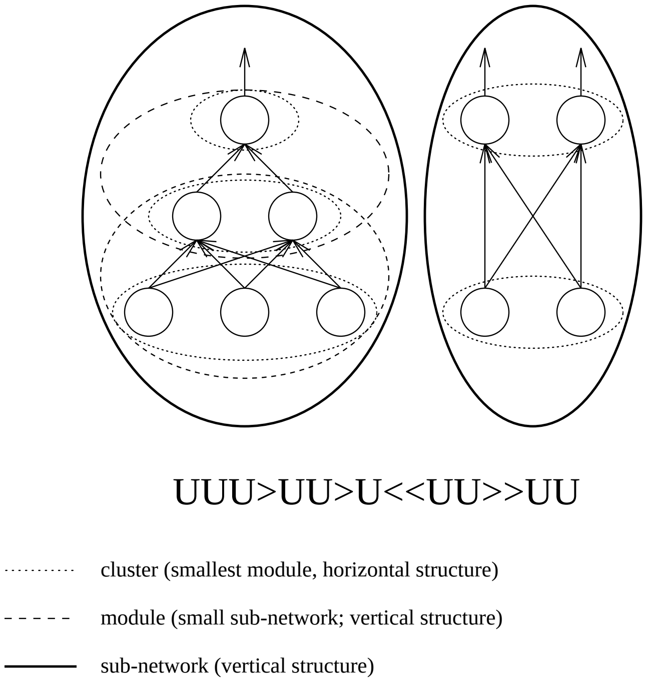

Figure 4.4: Visualization of how the short schemata criterion is met by the higher level genetic coding.

5. **compactness**
   The higher level genetic coding is not compact[13], as it is *not* non–isomorph. Nevertheless a fairly high compactness of individuals can be achieved by strongly restricting the growth of the individuals when setting parameters for the SCGA.

6. **non–isomorphism**
   The coding is *not* non–isomorph, because there is an infinite amount of different genotypes for each phenotype. This is due to the following facts:

   - The two operations ">" and "<" on the level register are inverse to each other. This results in the possibility that they may appear consecutively canceling each other without changing the topology encoded; e.g. the topology encoded with UU>U equals to the topology encoded with UU>><><><U.
   - The two operations ">" and "<" are meaningless, if they appear in the leading position of a genotype; e.g. >>><<UU>U equals to UU>U.
   - The operation "<" will be often ignored, if the level register value is 0; e.g. UU<<>U equals to UU>U.

[13]The lower level coding is also not compact, as it can e.g. contain empty translation table entries

Nevertheless the coding contains important features of a non–isomorphic coding. All units of a cluster are equal and the cluster is treated as a unity rather than as a group of individual units. This prevents identical clusters from being coded differently. The coding of clusters is non–isomorph. This principle would also be valid for modules and sub–networks, if not for the inverse nature of the letters “>” and “<”.

7. **modularity**
   There no modularity in the higher level genetic coding itself, but the coding is highly modular in connection with the lower level coding. Modularity is a feature of the lower level coding as the translation table provides a mechanism to encode clusters, modules and sub–networks and to reference them at different places in the genotypes of the lower level coding.

In conclusion it can be said that the unit–cluster model meets most of the quality criteria in an adequate way. The model however still lacks completeness, compactness and non–isomorphism. The compactness and also the non–isomorphism criterion, although not explicitly met, should have just a minor influence on the evolution, as they can be widely enforced by the growth restrictions applied using the SCGA[14]. How the very important completeness criterion can be satisfactory fulfilled will be show later. However, before a solution to this problem will be presented, there are some more advantages to the unit–cluster model that are worth being mentioned:

- The alphabet can be kept small and this leads to a straight forward model of topologies.
- The genotypes grow linearly[15] to the amount of units in a network, because single connections between units are not coded, as they are derived from the growing process.
- Relative unit positions are used and thus it is easy to define modular sub–networks (clusters, modules, sub–networks). The position of a modular sub–network in the neural network is then defined by the position of its genetic coding in the individual.

4.1.5 Extended Unit–Cluster Model

An extension of the unit–cluster model is necessary in order to meet the completeness criterion. The original unit–cluster model limits the topologies

[14] If the growth restrictions for growing genotypes are quite thight, then the genetic algorithm tries to find the best solution for quasi fixed size genotypes. If the size of the genotypes is sufficient small, then there soon is no place for redundancy in the coding.

[15] Actually the genotypes can even grow arbitrarily, due to the inverse nature of the letters > and <, but this can be suppressed by the SCGA.

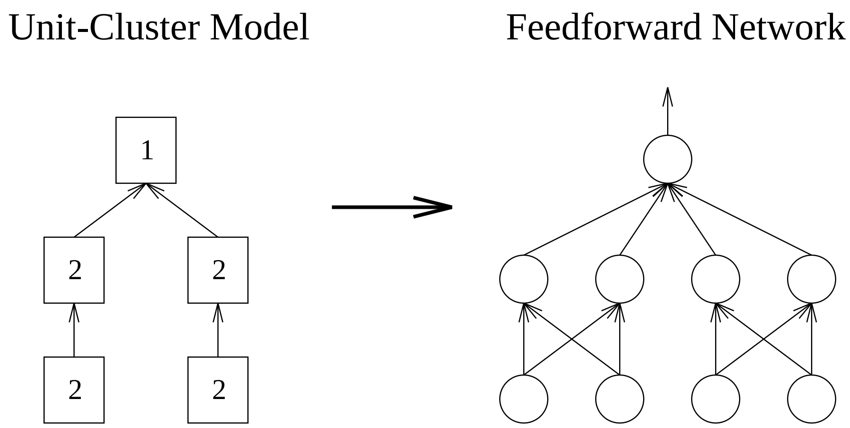

Figure 4.5: An example of a topology that cannot be encoded without an extension of the unit–cluster model.

to topologies that do not have clusters with fan-ins from more than one other cluster (see figure 4.5[16]). This however seems to be a quite serious limitation and therefore an extension had to be found.

The extended unit–cluster model (EU–CM) is established by adding a link register and two more letters to the alphabet. A link register is used to memorize the cluster to which a link can be established. A link is always made from the cluster currently created. Whenever a new cluster is being created the link register is reinitialized to point to the first but last cluster created that is at least one level below the cluster being created actually[17]. The two letters added to the alphabet use the link register. They belong to the cluster, behind which they appear on the genotype. They influence the fan-ins of that cluster and are therefore coded close to the cluster. The letters are "c" and "–" and they have the following meaning:

- **c**
  forces the connection to be made to the cluster to which the link register is pointing.

- –
  decrements the link register[18]. The link register is set to the cluster created earlier[19] (than the one it was pointing to) that is located on a lower level than the actual cluster. If such a cluster does not exist then no action is taken and the link register is not decremented further. If

[16] It is impossible to create this topology without an extension to the unit–cluster model, as in this case there is no mechanism that would allow the output unit (cluster) to be connected to all of the hidden units (two clusters).

[17] The cluster created last that is on a level below the actual cluster will be connected to the actual cluster in the growing process anyway as defined earlier.

[18] The link register is reinitialized for each cluster to point to the first but last cluster created earlier that is on a lower level.

[19] a cluster with a lower index

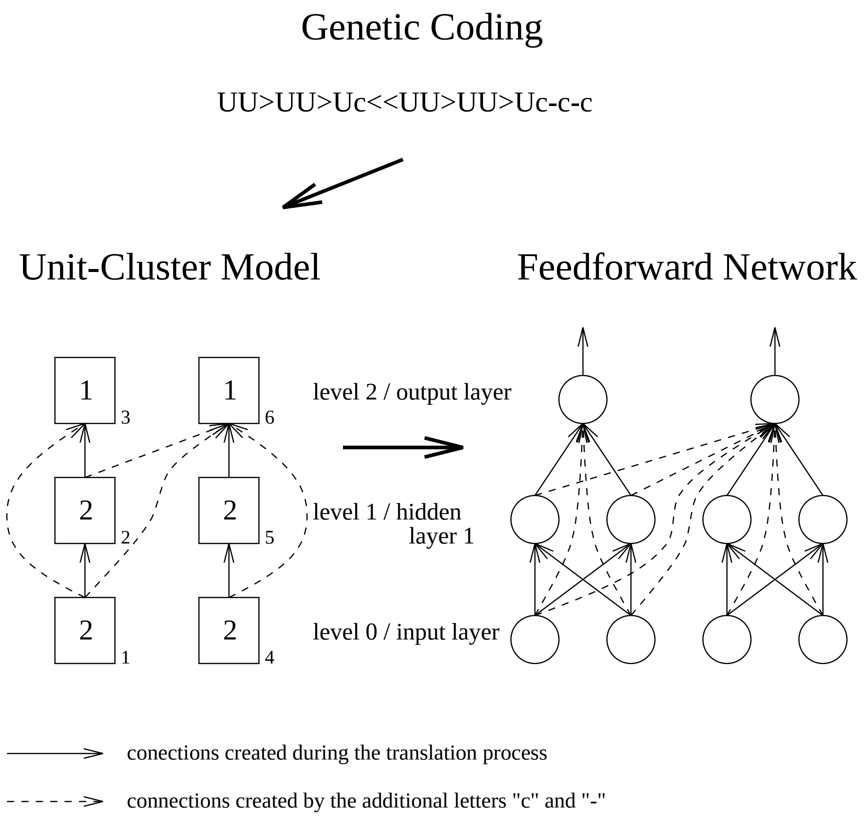

Figure 4.6: A topology using the extended unit–cluster model.

there is no cluster at all which meets the requirements[20] then the link register is set to “no links possible”.

Figure 4.6 presents an example of how the two additional letters are being used. The genetic coding is translated to the unit–cluster model first and then the feedforward network is directly derived from the unit–cluster model. The models without the dashed lines correspond to the individual without the newly introduced letters “c” and “–”. The decoding process depicted in figure 4.6 from the genetic coding to the unit–cluster model will be presented here in order to show how the newly introduced link register changes and how the additional letters take influence on the topology:

1. **initialization of the registers** →
   level register = 0
   link register = no links possible
2. **UU** →
   level register = 0
   link register = no links possible
   cluster 1 created

[20] E.g. the level counter is 0 and thus there is no cluster on a lower level.

3. $> \rightarrow$
   level register = 1
   link register = no links possible

4. **UU** $\rightarrow$
   level register = 1
   link register = no links possible
   cluster 2 created

5. $> \rightarrow$
   level register = 2
   link register = no links possible

6. **U** $\rightarrow$
   level register = 2
   link register = 1[21]
   cluster 3 created

7. **c** $\rightarrow$
   level register = 2
   link register = 1
   connect clusters 1 and 3[22]

8. $<< \rightarrow$
   level register = 0
   link register = 1

9. **UU** $\rightarrow$
   level register = 0
   link register = no links possible
   cluster 4 created

10. $> \rightarrow$
    level register = 1
    link register = no links possible

11. **UU** $\rightarrow$
    level register = 1
    link register = 1
    cluster 5 created

[21] Index of cluster to which a connection can be made.
[22] This letter **c** adds a fan-in to cluster 3 (it belongs to cluster 3). The established connection is a feedforward connection from cluster 1 to cluster 3.

12. $> \rightarrow$
    level register = 2
    link register = 1

13. **U** $\rightarrow$
    level register = 2
    link register = 4
    cluster 6 created

14. **c** $\rightarrow$
    level register = 2
    link register = 4
    connect clusters 4 and 6

15. $- \rightarrow$
    level register = 2
    link register = 2

16. **c** $\rightarrow$
    level register = 2
    link register = 2
    connect clusters 2 and 6

17. $- \rightarrow$
    level register = 2
    link register = 1

18. **c** $\rightarrow$
    level register = 2
    link register = 1
    connect clusters 1 and 6

Finally, the genetic coding is complete, as all possible topologies of feedforward networks can be encoded onto genotypes[23]. A formal prove will not be given here as the matter is trivial. Once it is possible to connect every cluster to every cluster that is on a lower level or not to connect them and given that the clusters may have the size of one unit then all connectivity patterns are possible.

4.1.6 Restricted Unit–Cluster Model

The extended unit–cluster model (EU–CM) satisfies in connection with the SCGA the demands put up to the genetic coding of feedforward neural networks quite well. Nevertheless another important demand can be found that should be incorporated. This demand is of rather practical nature.

[23]Note that the transition function is here always a sigmoid and that generally only feedforward networks with *one* transition function can be represented using this alphabet.

Usually, when there is a need for a feedforward network the input and output data structures are known and usually all inputs and outputs are being used[24]. When the amount of input and output units is known, it is a priori knowledge that should be used. It restricts the solution space and therefore it saves time to make this information available to the genetic engine[25]. There are basically two possibilities to make this knowledge available to the genetic engine:

1. *Encode this knowledge in the fitness function.*
   This possibility could be easily enforced by giving a strong quality penalty to individuals that do not have the appropriate amount of input and output units[26]. Nevertheless the genetic algorithm would need some time to evolve such networks and secondly there would always be a chance that during the course of evolution lethal individuals would be created[27] that would use costly resources like computational time.

2. *Define the amount of input and output units and develop just the topology of the hidden units genetically.*
   This approach is better, as it allows to set the input and output layers manually. Thus the genetic engine just has to find the topology of the hidden units. Known parts of the final solution are set manually and this saves time.

The second approach was chosen. The input and output layer are predefined and the extended unit–cluster model is used to develop the topology of the hidden units. After decoding a genotype, all units with no fan-in are connected completely to the input layer (instead of being treated as input units) and all units with no fanout are connected completely to the output layer (instead of being treated as output units). Figure 4.7 shows an example. Furthermore the meaning of the letter "–" is slightly adjusted so that links to the input layer can be established. From the point of view of the link register the input layer is defined to be the first cluster. Thus the link register can be decremented further (exactly by one) to enable links from hidden units to the input layer.

This model unfortunately limits the completeness criterion. Nevertheless the test work done in this research was limited to this version in order to focus

[24]Unless there is redundancy in the input or output data so that some input or output values (units) are not needed.

[25]The genetic algorithm does not have to find structures with a certain amount of input and output units with no guidance and thus computational time is saved.

[26]The penalty should be of magnitudes larger than the actual quality in order to force the genetic algorithm to find networks with the right amount of input and output units before optimizing the rest of the topology. Furthermore, the penalty should be decreasing the closer the desired amount of input and output units is met by the individual so that a gradual adaption is possible.

[27]All individuals that would not have the demanded amount of input and output units would be lethal, as they would have no chance of survival (although they might be perfectly correct feedforward networks).

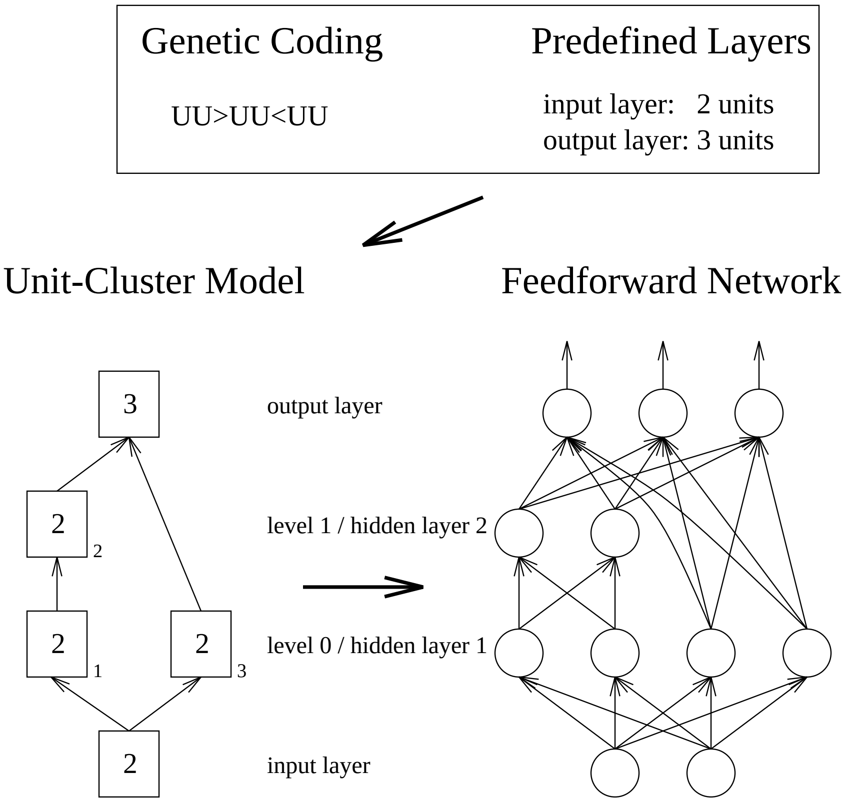

Figure 4.7: A topology encoded using the restricted unit–cluster model with predefined input and output layers.

on the performance of the restricted unit–cluster model in its simple form. Once it can be shown that this restricted unit–cluster model can be used by the genetic engine to find good solutions for certain problems that do not need a further extension of the model, it should be extended to meet the completeness criterion and to be able to solve a wider variety of problems. The completeness criterion is violated as there are two feedforward network classes that cannot be coded genetically:

- The input and output layers are treated as one cluster and thus there is no possibility to create links from one cluster to just a part of the input or output layer.
- There is no possibility to create shortcuts to the output layer. There is no mechanism to connect clusters to the output layer except when they are connected directly, meaning that after the decoding process they have no fan-outs and are automatically linked to the output layer.

No solution to the first problem has been elaborated yet that would not deform the idea of the unit–cluster model. A special mechanism has to be found in order to incorporate modularity of the input and output layers in a way that would fit the unit–cluster model. It is not sufficient to treat every unit of the input layer as a cluster of its own, although this would be an approach that could lead to a solution of the problem.

The second limitation however could be removed easily by introducing another letter to the alphabet, e.g. "+". Whenever a "+" would appear after the coding of a cluster in the genotype this cluster would be connected to the output layer with a shortcut. The problem of creating a shortcut from input to output layer would still remain, but could also be easily solved by adding a letter to the alphabet, e.g. "I". Whenever this letter would appear at least once in the genotype a shortcut from input to output layer would be made. These suggestions have not been tested yet, but seem to fit the idea behind the general unit–cluster model.

A final limitation, which is just a part of the second one, but should be mentioned by itself, is the fact that whenever there is no hidden unit at all the input layer is *not* connected to the output layer. This results in a lethal individual, but since the emphasis of this work is on the unit–cluster model and its modularity features, there is no desire to test this approach on problems that can be solved by networks with no hidden layer[28].

4.2 Fitness Functions

The development of an appropriate fitness function is a very important part of the design of an evolutionary neural network constructor. The fitness function guides the genetic algorithm and takes therefore a very strong influence on the solution. The selection of the right fitness function strongly depends on the individual problem, for which a solution is being searched. Nevertheless there are certain basics that apply to most[29] fitness functions. The purpose of this section is to elaborate several fitness measures and discuss their influence on the genetic search.

If the genetic coding of the feedforward networks had been a DES coding in its most strict form then the task of finding an appropriate fitness function would be reduced to finding a set of test data and the fitness of each network could be calculated by measuring its performance on the test data. This however is only possible, if the network is coded completely with all its weights and biases. The IES coding used here requires more effort.

The network topology by itself is not sufficient to provide a good quality measure, although it contains some information that can be used and will be explicitly mentioned later. In order to turn the topology into a network that can be evaluated by applying a test set of data, a learning algorithm

[28]There are few problems that can be easily solved by a network with no hidden layers, while a solution with hidden layers is rather difficult to find, but these problems are rather rare. An example can be found in the thesis of Boers and Kuiper [1]. It is based on the the "what" and "where" categorization of Rueckl et al. [27].

[29]probably even all

has to be applied. There exist several learning algorithms for feedforward networks – e.g. backpropagation [28, 29], conjugate gradient [13, 24], simulated annealing [8], genetic and evolutionary approaches [21, 33]. The choice of the learning algorithm with which the network is then trained has a major impact on the topologies being evolved during the evolutionary process. The goal of the genetic engine changes from finding just a network that will solve the problem to finding a network that can be trained by the chosen learn algorithm to solve the problem. This however limits the amount of solutions that can be found.

In this study the backpropagation algorithm was chosen to be the learn algorithm that was used to train the networks and deliver a fitness measure. The backpropagation algorithm was selected for two major reasons:

1. The backpropagation algorithm implements a local gradient descent search and is therefore much faster in finding a minimum than a global search technique like simulated annealing or genetic and evolutionary approaches. Of course, the backpropagation algorithm is only able to find a local minimum, but nothing more is needed, as the topology influences the shape of the error surface of the problem. It is possible and desirable to find topologies that shape the error surface to be unimodal.

2. The backpropagation algorithm is highly general applicable, meaning that it is possible to train all possible feedforward networks using this tool[30]

The conjugate gradient algorithm could have been chosen as well, as it also meets the two criteria stated above.

4.2.1 Single Fitness Measures

With the evaluation algorithm selected it is possible to list the different fitness measures and discuss their applications and combinations. Different measures that can be included in a fitness function are:

1. amount of units (topology definition / genetic coding)

2. amount of connections (topology definition / genetic coding)

3. train error (backpropagation)

4. test error (backpropagation)

[30]The networks will learn, if they are capable of capturing the function behind the data. Otherwise they will not learn, but they still can be trained.

Those measures can be used separately or can be combined with each other. A close up will be taken later in order to show which combinations can be useful and where there are problems. First however, those measures need to be analyzed in order to check their importance and their possible application:

1.) **amount of units**
The usage of this measure in a fitness function is limited and it cannot be used by itself[31]. Nevertheless it can be used to guide the genetic algorithm to develop the correct amount of input and output units. This is not interesting for the restricted unit–cluster model, as the input and output layers are predefined, but very useful, if applying e.g. the extended unit–cluster model. If this measure is used for the purpose of finding the input and output layers, then it should be given a strong influence on the total fitness measure in order to force the genetic algorithm to find correct input and output layers quickly.
Another possibility is to use this measure for defining a range where the genetic algorithm should search for the amount of hidden units needed to solve the problem. Individuals that would have too many or too few hidden units should be given a fitness penalty in order to force the genetic algorithm to search just inside the predefined range. This would narrow the search space, but requires an a priori knowledge that is usually not given.
It is often desirable to find networks with a minimal amount of hidden units that solve a problem, as those networks tend to generalize better than larger networks. This however cannot be forced by using this measure, because one runs into a paradox. The paradox is that once the exact amount of hidden units is known one does not need to search for it using the genetic algorithm, but it has to be known in order to make it an influential parameter in the fitness function.

2.) **amount of connections**
The amount of connections is also not a stand alone fitness measure for similar reasons as the amount of units is not. It is even more difficult to incorporate this measure in a fitness function than the amount of units. One possibility however is to define a search range like for the amount of units and give fitness penalties to individuals with more and less connections. Again the usage of this measure in a fitness function requires some a priori knowledge.

3.) **train error**
The train error of the backpropagation algorithm measures the capability of the neural network to learn the data presented or rather to learn a function[32] that will fit the presented training data best.
This measure may be used as a stand alone fitness function and its position

[31] There is no possibility to see the performance of a feedforward neural network by just looking at the amount of units it has. Except for trivial cases like a network with the wrong amount of input or output units or a network with obviously too little or to many hidden units (provided the user has some a priori knowledge of the problem).

[32] not necessary the one desired by the user

in a topology search setup will be mentioned later. There is usually no desire for a neural network to just memorize data as this can be mostly done more efficiently by a database[33]. Nevertheless a network must first be able to learn the training data before it can generalize and therefore the train error may be quite useful during topology search.
The train error of different networks can be compared in several different ways:

- *compare the train error after a certain amount of epochs*
  This method compares two networks by measuring their learn performance after they have been trained with the training data set (the training data set has been presented to them) a certain amount of times (epochs). This measure is close to measuring the error after a certain amount of training time has elapsed in a parallel hardware implementation of the network. Furthermore, this measure is highly portable as it does not depend neither on the hardware used nor on the implementation of the backpropagation algorithm.

- *compare the train error after a certain time has elapsed (serial implementation)*
  This measure is of importance just if there is a need to consider that the neural networks are simulated on serial machines (e.g. von–Neumann computers). Larger network need more time to be trained than smaller networks on such computers. This gives an advantage to smaller networks, as they will be trained more often on the training data set than the larger ones within the same time. This is usually a nice side feature, but has little to do with the idea of neural networks as a structure.

- *compare the train error after a certain time has elapsed (parallel implementation)*
  This measure is not much different from the first one. It only gives an advantage to networks with less layers. The training data set can be again presented to those networks more often, than to networks with many hidden layers.

The train error after a certain amount of epochs has been chosen for this study. It seems to be a better measure than the train error after a certain time has elapsed in the serial version and there seems to be just a small difference to the train error after a certain time has elapsed in the parallel version[34]. Furthermore this measure is extremely attractive, because of its high portability.

4.) **test error**
The test error measures the ability of the network to generalize. It can be

[33] except maybe, when using the network as an associative memory

[34] It would be however interesting to investigate the influence of this difference. Networks with fewer layers would have a great advantage and it would be interesting to see, whether multi-layer networks would still develop.

used as a stand alone measure. The advantage of using the test error instead of using the train error is that the networks are limited in their growth as networks that contain too many free parameters[35] usually perform worse than smaller ones. Furthermore the networks are being directly tested for the task they have been designed to do – generalize a solution. The problem with using this measure however is that the user has to make sure that the networks are not permanently overlearning[36] the problem by adjusting the amount of epochs during which they are trained. Thus the test error could be used in connection with the train error.

4.2.2 Combined Fitness Measures

The different fitness measures can also be combined to a fitness function. A combination could e.g. be the sum of all measures. Such a combination always has to be created with respect to the different scopes of the single measures. The train error and the test error may have different scopes and the amount of units and connections have almost always different scopes. It is usually not trivial to find the right coefficients for the single measures in order to weight them in a desired way within the sum. Sometimes the sum might not be a sufficient way of combining them and a polynomial has to be used. A possible general form could look as follows:

$$fitness =$$
$$a_0 \cdot (\sharp units)^{n_0} + a_1 \cdot (\sharp connections)^{n_1} + a_2 \cdot (trainerror)^{n_2} + a_3 \cdot (testerror)^{n_3}$$

The difficulty lies in establishing good values for a_i and n_i. Those values are mostly dependent on the problem and the desired solution.

Examples of combined fitness functions could e.g. be:

- *combination of train error and amount of hidden units and connections:*
  If the networks are not supposed to just memorize the training data then networks that exceed a certain size[37] could be penalized. The fitness penalty should be significant and quickly increasing once the networks become too large.

- *combination of train and test error*
  The test error could be combined with the train error in order to avoid problems with overlearning. If just the test error is used, then a network that has been overlearned might have a lower fitness than a network that cannot generalize that well at all.

[35] Free parameters are weighted connections and biases.

[36] overlearning – fitting the training data too well so that the approximation is not general any more. It may happen when a network has been trained for too many epochs. The result is a network that cannot generalize, although it fits the training data set extremely well.

[37] e.g. have more free parameters (weights and biases) than training examples

There are many possibilities to define a fitness function and the choice always depends on the problem to be solved. A possible setup for searching a topology with different fitness functions will be presented here. The goal is to find a network that will be able to perform a good generalization after being trained with the training data set:

1. *Set the fitness function to be the train error of the backpropagation algorithm*[38]*.*
   The analysis of this optimization run should reveal the size of networks that should be capable of learning the training data set.

2. *Set the fitness function to include the train error and the amount of connections and units.*
   The amount of connections and units equals to the amount of free parameters (weights and biases) of a network. With this measure it is possible to narrow the search space to networks that are large enough to capture the problem, but do not grow beyond a predefined size[39]. There is not much sense in searching through network topologies that have by far more free parameters than training data examples[40]. This limitation is of great advantage as larger networks use more time during training on sequential computers than smaller networks. By limiting the network size the time saved can e.g. be spend on letting the networks learn for a longer time[41]. Furthermore a greater chance is given to the development of a certain structure, because when the networks cannot grow further, their performance can only improve by modifying their topology till an optimal structure is found.

3. *Set the fitness function to be the test error or the combination of test and train error.*
   If during the evolution with at least the train error as the fitness function progress is made, then the evolutionary run should be repeated with the fitness function set to the test error or even better to a combination of the train and test error. After the previous two evolutionary runs a promising starting point for the genetic engine should be easy to find[42]. It must be noted that solutions found using this fitness mea-

[38]Train the network till a certain amount of epochs. To find out the amount of epochs needed it is often useful to construct a network, which is too large, and see how many epochs it takes for the train error to decrease significantly. The network should have far more free parameters (connections and biases) than training examples (data).

[39]With the mere train error as the fitness this is quite likely to happen.

[40]Best is to have at least twice as many examples as free parameters of the network in order to assure that generalization can develop.

[41]As the training of the networks is computationally highly expensive the networks are usually not completely trained, but just trained till they show a significant behavior. This however creates a learning horizon. It is quite possible that some networks will learn well in the beginning, but will still be unable to capture the entire problem. Thus it is of great advantage to let them learn as long as possible (or till they capture the entire problem).

[42]Networks of certain sizes that have been observed during the previous evolutionary runs could be selected for the initial population.

sure should be tested with a validation data set after the evolutionary run. The validation set is a third data set that has not been used during the evolutionary process. This test is necessary as it is possible that networks have been found that are capable of fitting the test set when trained with the training set, but still do not have the desired generalization features. An out of sample data set is necessary to test the generalization abilities of the networks.

One could ask why this setup should be used instead of just using the last fitness function. It does not have to be used, but it enlarges the chances of finding a good solution even for more difficult problems. The test error often decreases slower than the train error and demands therefore from the network that it learns a greater part of the desired function. Thus it is more sensitive to the right size of the network and networks which are e.g. too small might produce a test error that is of smaller scope than the noise introduced by the random weight initialization when using the backpropagation algorithm. It is therefore desirable to find networks that can at least learn the training data first, as only those networks may also develop the ability to generalize. Furthermore as the test error usually shows a significant decrease after more epochs than the pure train error, time could be wasted if the range of network sizes was not found earlier.

It is also possible to change the fitness functions during just one evolutionary run. Thus it might be even better to start the genetic search with the train error as the fitness function, change it after good solutions have been found to the train error with a network size limitation and finally set the fitness function to the test error (or test and train error). Some adjustments to the amount of training epochs might be useful in this case[43].

In the experiments made in this study the fitness function has been set to be the train or test error. This, of course, covers only a part of the suggested way of applying the different fitness functions, but further experiments would have exceeded the time limits of this work. Nevertheless the analysis of those tests reveals many interesting aspects of topology optimization with genetic algorithms, the restricted unit–cluster model and the semantic changing genetic algorithm.

Before those tests will be presented, it should be mentioned that further fitness measures than the ones presented here can be found, but this would go beyond the scope of this work.

4.3 Tests

In this section tests with several different problems are presented and analyzed in order to show the capabilities and limits of the semantic changing

[43]The amount of training epochs can be e.g. increased for the second stage of the evolution and decreased for the third stage.

genetic algorithm applied to the restricted unit–cluster model. The problems were selected to include binary and real valued input and output data for the neural networks. Four tests were conducted – two tests with synthetic data and two with real world data. The problems tested with synthetic data are the 6–bit integer addition and the partition problem. The real world problems are a breast cancer prediction and a profile construction for plane engines. This divers selection is supposed to give some insight to the behavior of the genetic system and its solution strategies. It should be noted that only a few test could be made due to computational and time limitations. The results presented here are representative results for usually two independent runs. Before however, the tests are presented some thought is given to the setting of parameters for the SCGA and the backpropagation algorithm (BP).

4.3.1 Setting Parameters

This section covers the most important parameters of the SCGA and the BP that have to be set by the user. Most of the parameters were set once and left unchanged for all experiments. This was done in order to check the robustness[44] of the genetic system with respect to parameter settings. Better parameters could probably be found, if they were adjusted for each single problem individually. If this however had turned out to be a necessity, then the usability of the genetic system would be limited, as tuning all parameters for each problem is not a simple task.

Parameters for the lower level coding and the genetic algorithm:

- **Population Size = 20**
  The population size was set to 20, because:

  1. Smaller populations showed to not evolve solutions quickly or at all.
  2. Larger populations did not seem to improve the speed of convergence of the genetic system significantly[45].
  3. Due to computational limitations the population had to be kept small.

- **Length of Individuals (Lower Level Coding) = 60 (120)**
  The length of the bit-string coding is highly dependent on the size of the translation table. The goal was to start out with higher level coded

[44]The robustness of an algorithm with respect to the parameter setting is a measure for how important it is for the algorithm to have exactly adjusted parameters in order to find a good solution. The less accurate the parameters have to be set the more robust is the algorithm.

[45]Populations with up to 100 individuals were tested. If however there is the possibility to use much larger populations, (e.g. 1000 individuals or more) it might make a difference. This could not be tested due to computational limitations.

individuals that would consist of 10 (20) symbols (max. 10 (20) hidden units in the first generation). 6 Bits were decoded to a translation table entry, which in the beginning contained one symbol, and thus 60 (120) bits were decoded to 10 (20) symbols. There are several advantages in starting out with small networks:

1. With less table entries accessed by an individual the table entries are more likely to encode more interesting structures (clusters, modules, sub–networks), as larger structures need to be encoded in one table entry when the networks grow in size.
2. Starting out with small networks makes it easier to see the development of the solution over the generations. This, of course, is only important when testing the features of the genetic system. If the goal is to find a solution as quickly as possible and it is known that a certain network size is at least needed, one should begin the evolution with networks of that size.
3. Finally, smaller Networks require less computational time to be evaluated (the simulation was done on a sequential computer). Thus it is computationally cheaper to grow networks rather than to start optimizing with networks, which are too large and let the genetic system shrink them.

For the partition test this parameter was set to 120. Larger individuals were created in the first generation with the hope that a solution would evolve quicker.

- **Crossover Rate = 0.8**
  This is a standard value for the crossover rate as often found in literature.

- **Mutation Rate = 0.05**
  As mutation on the lower level coding[46] is a highly disruptive operator[47] it has to be set to a low value.

- **Generation Gap = 0.9**
  The generation gap defines how many percent of the old generation are replaced by the new generation. In this case here just 10% of the old generation survives. However, the individuals who survive are chosen arbitrarily. Thus a high value for the generation gap should be chosen.

- **Elitist Strategy**
  The elitist strategy has been chosen in order to increase the chances for the best individual to advance to the next generation. The best lower level coded individual of the population automatically advances to the next generation. Setting this parameter increases the chances

[46] bit–flip in the bit-string individual

[47] It is highly disruptive, as it changes the higher level coded individual by an entire table entry.

of the best solution to survive, but it does *not* assure it. During table adaption the best individual can be changed, if a table entry that is accessed by the best individual is changed. This however, is for really outstanding individuals very unlikely to happen, provided a fitness function is chosen that will honor outstanding performance in an outstanding way (see section 4.3.2).

Parameters for the higher level coding and the semantic changing extension of the genetic algorithm:

- **Alphabet = {U, >, <, c, -}**
  The alphabet of the extended unit–cluster model was used in all tests for initializing the translation table and mutating chosen table entries. However, the distribution of the letters in the initial table was different for the single letters. The letter U occupied 40% of the initial translation table, the letters > and < 20% each and the letters c and - 10% each. This was done, because the letters are of different importance. The U is needed to create units and is therefore most important, because when there are no units, no structure can be evolved and the other symbols become meaningless. The symbols > and < are second on the importance scale as they are needed to create the main modular structure. Without them again the letters c and - become meaningless. Although, of course, the SCGA is able to modify the translation table and rate the importance of the letters by itself, the percentages were designed to aid the adaption and save time.

- **Translation Table Size = 2^6 = 64**
  The size of the translation table depends on the length of the lower level coded individuals and the population size. A table that has many table entries that are not accessed by any individual[48] slows down or even disables the work of the semantic changing extension of the genetic algorithm and is too large. A table that has only few entries is too small, if the top individuals of the population are permanently destroyed by table adaption[49]. The table size of 64 seemed to be a good choice for a population of 20 individuals of which each individual accesses 10 table entries. It allows different amounts of table spaces to be modified, without always killing the top individuals and without disabling the semantic changing extension of the genetic algorithm.

- **Percent of Table Spaces that are modified each Generation = 50%**

[48] here: (population size $\cdot$ length of initial higher level coded individuals) = max. accessed table spaces; In the tests presented here 200 table entries were maximally accessed. A table with e.g. 1000 table entries would probably be too large.

[49] Each individual in this study accessed 10 table entries. Experiments show that it is advisable to leave at least double as many table entries unchanged during table adaption as are maximally accessed by one individual. This prevents the table adaption from destroying the top individuals.

An individual accesses, with respect to the setting of the other parameters, at most 10 table entries. The table has 64 table entries. As mentioned earlier, it is advisable to leave at least 20 table spaces unchanged. The decision to leave 32 table entries unchanged was guided by the goal to preserve the best solutions with a high probability. Leaving even more table spaces unchanged however, would diminish the influence of the semantic changing extension of the genetic algorithm unnecessarily.

- **Mutation Rate for the chosen Table Entries during Table Mutation = 10%**
  This table mutation rate sets the probability, with which a chosen module of a donor individual is mutated before it is inserted in the translation table. Each letter of the module can be exchanged with a random letter of the alphabet with this probability. The rate is set quite high for two reasons:

  1. Individuals that are mostly affected by the table mutation are of a lower quality and therefore this gives room for experimenting. Even if the mutation destroys a good module, the module is still likely to survive in one of the individuals that have a high quality.
  2. Once a letter is abandoned from the translation table, it has a greater chance to enter the translation table again[50].

- **Gaussian Maximum = 10% (5%)**
  The Gaussian maximum defines the length of the modules chosen from the donor individuals during table adaption that is most likely selected. The length is defined through a relative length of the donor individual. The Gaussian maximum was always chosen to most likely keep the average length of the higher level individuals constant. The goal was to let the genetic engine search for a good solution without forcing it to grow or shrink the individuals. To calculate the Gaussian maximum the average length of the higher coded individuals and the average length of the initial table entries has to be regarded. In all of the experiments (except the partition test) the average length of the individuals was 10 letters (partition test: 20 letters). The average length of a table entry of the initial translation table was 1, as the table was initialized with letters of the alphabet. In order not to force the table entries to grow or shrink, the most likely chosen length of the modules that should enter the table during table adaption should be initially 1 and this is exactly 10% (5% for the partition test) of the length of an initial individual. Once this parameter is calculated in the described

[50] Experiments showed that the genetic engine often disables the "level up" feature (letter ">") in the first part of the evolution by kicking it out of the translation table during table adaption. When the networks reach a certain size however, this letter is reintroduced by this mutation and it turns out to be very useful in order to improve the quality of the networks.

way the non-growing and non-shrinking feature holds even when the individuals change in size during table adaption.

- **Gaussian Variance = 1%**
  The Gaussian variance defines how strongly the prefered length of modules that enter the translation table is enforced. The smaller the variance the less likely the size of the translation table entries will change. A small variance makes it unlikely that the table entries will change in size[51], while a large variance makes a change highly probable. 1% turned out to be a good value, as it hinders the individuals from a rapid growth, but does not make it almost impossible. It lets the genetic algorithm search for solutions locally[52] growing and shrinking the individuals in a moderate way.

Parameters for the backpropagation algorithm (for an exact description of their influence on the training of neural networks see [29]):

- **Amount of Epochs**
  The amount of epochs sets how often the training examples are presented to the neural networks before their training and test error are reported. This parameter has to be set individually for each test and will therefore be mentioned later. However there is a common strategy behind the setting of this parameter for each problem. The parameter should be set large enough so that the different networks can show a significant behavior. A possibility to gain an idea of how to set this parameter is to try training networks that are by far too large to be good solutions, but will learn the problem.

- **Learn Rate = 0.1**
  The learn rate defines how strongly the weights of the neural network are adjusted during each training cycle. The learn rate was kept small, as a small learn rate is likely to result in a smoother descent to the local minimum than when a larger learn rate is used. The smaller the learn rate is however, the slower is the descent and the more computational time is needed[53].

- **Momentum = 0.9**
  The momentum term defines the influence of the descent direction of the previous epoch. The term is not necessarily needed, but it sometimes speeds up the descent. Similar to the learn rate this parameter is also problem dependent.

- **Weight Range = 0.1**
  The weight range sets the interval for the random weight initialization

[51] Provided that the Gaussian maximum is *not* set to force the individuals to grow or shrink.
[52] on individuals of similar sizes
[53] Learn rate and momentum are parameters that are highly problem dependent [29].

of the networks. In this case here the weights of the networks are randomly initialized with values from the interval [-0.1, 0.1] before the networks are being trained. It is common to initialize the weights with values from the interval [-0.5, 0.5]. The interval was chosen smaller in order to minimize the influence of the random initialization on the performance of the networks. As the networks were only trained for a short time, the larger interval turned out to submit too much noise to the fitness function in an early stage of the evolution.

4.3.2 Fitness Functions

The tests conducted in this research were done using just single fitness measures. Although additional fitness measures and a fitness schedule have been suggested in section 4.2, only two fitness measures could be tested due to time limitations.

The two fitness measures that were used separately in different runs were the squared train error (tss^2) and the squared test error (tss^2). Using the train error is supposed to evolve networks that will learn the problem. The test error should force the SCGA to create networks that will have good generalization capabilities. The networks were *not* trained till they captured the problem completely, but just for a short time (small amount of epochs). This could lead to a possible learning horizon[54]. The total sum of squares (tss) was squared again, in order to stronger rate the differences between the networks and thus to save outstanding solutions from being lost during table adaption[55].

Finally, it should be mentioned that the fitness was always a positive real value with the best fitness being 0.0. This differs from the idea of fitness as presented in chapter 3, as there greater values indicated better fitness. It is however just a matter of definition and has no influence on the genetic algorithm.

4.3.3 6–Bit Addition Test

The 6–bit addition problem consists of finding a network that will be able to add two six bit binary coded integers[56]. The result is a seven bit binary coded integer.

Two fitness functions were used separately and two tests were conducted for each fitness function. The fitness functions were defined by the squared

[54] The learning horizon – if a network is not trained till it captures the problem completely, one cannot be sure that it is capable of learning/generalizing it completely. The only fact known is that it can learn/generalize a part of the problem till a horizon.

[55] Outstanding individuals might access table entries that are not accessed by any other individuals yet. Those table entries should be saved and not overwritten. The tss by itself however, turned out to be insufficient to protect those table entries.

[56] This should, of course, work for all six bit integers.

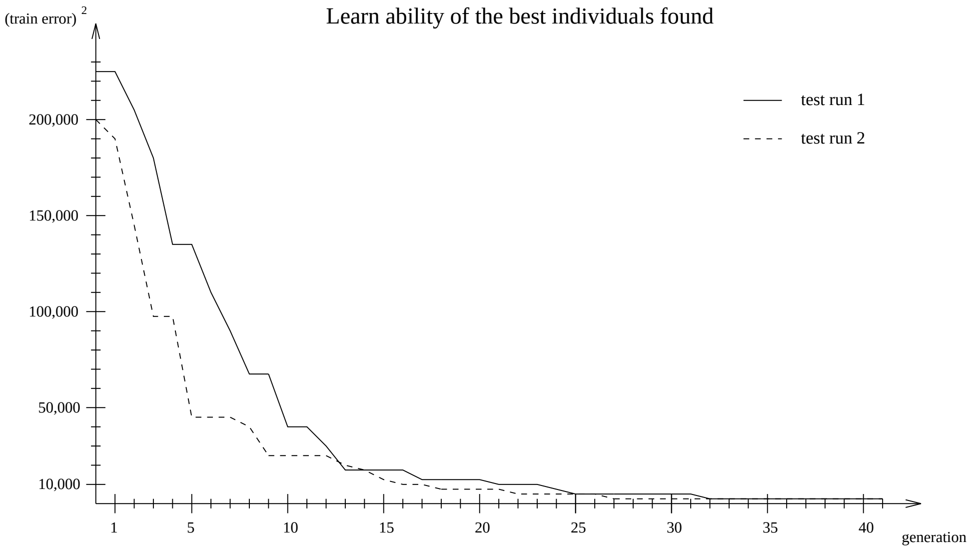

Figure 4.8: Fitness of the best individuals for learning the 6–bit addition.

train and squared test error.

Each individual (network) was trained for 40 bp–epochs on a training data set of 500 randomly selected input-/output patterns. Then the test error was taken to define the fitness function. The trained networks were presented a test data set of all possible 4096 patterns in order to check their generalization ability.

Train Error

Taking the train error to define the fitness function aims at developing networks that can be quickly trained to learn the training data.

The point of convergence[57] in this test was reached in the two independent runs at generation 27 and 32. Figure 4.8 shows the qualities of the best individuals during the evolution. The graph shows that the genetic algorithm is capable of quickly evolving networks that learn the 6–bit addition in a short time. The average fitness of the population improved in a similar way, although slower.

In order to see the development of the structures, best structures evolved in run 1 are denoted in figure 4.9[58]. These are structures that have been evolved in generations, where a strong advance of fitness could be observed (see figure 4.8). The higher level genetic coding of the best individual of

[57]A point in the evolution after which there is not much change in the best fitness found later.

[58]The individuals depicted do not show the complete modular structure coded in the genetic coding. Units that belonged to one cluster, but were coded in two clusters, were collected in one cluster. E.g. all units on the first hidden layer that have fan–outs to the output layer.

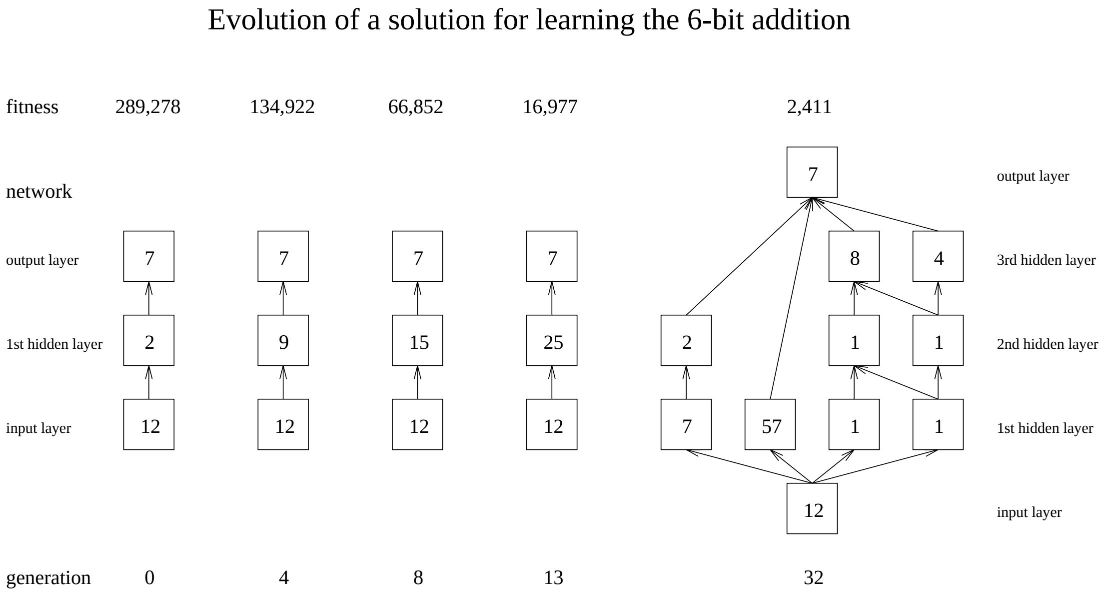

Figure 4.9: The evolution of topologies for learning the 6–bit addition.

generation 32 was:
c<UUU<<cUUUcU<UUU<UcUU<c-UUUUU>UU<cUUUUcUcU>-
<UU-<U<UcU>-<UU-UcU<UcUc<UU<UUUU<<UUUUUU-U-UcU<-
<<UU<UcU<U-<U>UcU>UUcUUcU-UUUcUUU<>U<<c->-
<cUUUUcUcU>-<UU-UU
It should be noted that in the evolution of run 2 similar structures were evolved.

The SCGA first grows a single layer and is able to improve the fitness of the individuals significantly. The genetic algorithm discovers that it can improve the fitness of the networks faster by growing the amount of hidden units in the first hidden layer instead of creating multilayered structures. This is probably due to the fact that the SCGA limits the speed with which the individuals can grow and thus searches the best solution localy without imediately growing the individuals. The individuals of the first generation consist of 10 letters. The SCGA favours evolving individuals that contain only the letter “U”[59] instead of decreasing the amount of “U’s” in the individuals by letting them contain structural information (other letters). Another evidence that supports this theory is the fact that the crucial symbol for creating multilayered structures (“>”) has vanished from the translation table of run 1 early in the evolution (generation 4). Its occurence was also strongly diminished in the translation table of run 2 during the first generations. In generation 4 it existed in only three table entries and this although it occupied 20% of the initial translation table. This symbol was reintroduced in later generations during the mutation part of the table adaption.

[59] greatest amount of units

Although the fitness of the networks could be improved significantly by creating one large hidden layer, the solution of both runs contain multilayered structures. This hints at the interpretation that after a certain point in evolution multilayered structures are needed to further improve the learning capabilities of the networks. The solutions of the last generations are similar to each other in the way that they are often networks that contain three sub–networks – one large single layered sub–network and two small multilayered sub–networks with two or three layers. The question however, whether the multilayered sub–networks are really needed to further improve the fitness of the networks or whether the growing single layered sub–network is sufficient, remains open.

When it comes to the genetic coding of the best individual of generation 32, it can be seen that it contains many redundant letters (<, c, -). All the < letters that are decoded, while the level register value is 0 are mostly ignored and the same also holds for the letters c and -. Those letters contribute structural information that is not used. This sleeping genetic material might be used when the sub–strings are used in a different context (e.g. greater level register value). On the other hand, they do not harm the individual, except for making the coding not compact and leading to unexpected side effects when being used in a different context[60].

Test Error

The described tests were repeated, but this time the fitness function was defined by the test error in order to generate networks that would be able to generalize the learned data.

In this test the point of convergence was reached in generation 38 in the first run and in generation 43 in the second run as depicted in figure 4.10. This tests shows that the SCGA is also capable of developing networks that have the ability to generalize, when learning the 6–bit addition.

In order to analyse the behaviour of the SCGA during the evolution, a few best individuals are depicted in figure 4.11. The individuals are taken from both runs and reflect the gradual development of the individuals at the crucial points in the evolution.

The best individual that has been evolved in run 1 and is depicted in figure 4.11 has the following genetic coding:
U-UUcUUU<U<Uc-U<UUcUUUUUUUU>-UUU-UU-cUUUUcUUUUUU
cUUUUUUUU<UU<UUU><UUUUU<UUUU<Uc>UUUc<UUUU

In comparison to the test runs where the fitness function was defined by the train error the SCGA favors rather multi layered structures here. Again

60 The unexpected side effects occur, as a sub–string like e.g. UU<UU is regarded as being equal to UUUU when the level register value is 0 and it might have a different meaning when the level register value is greater than 0.

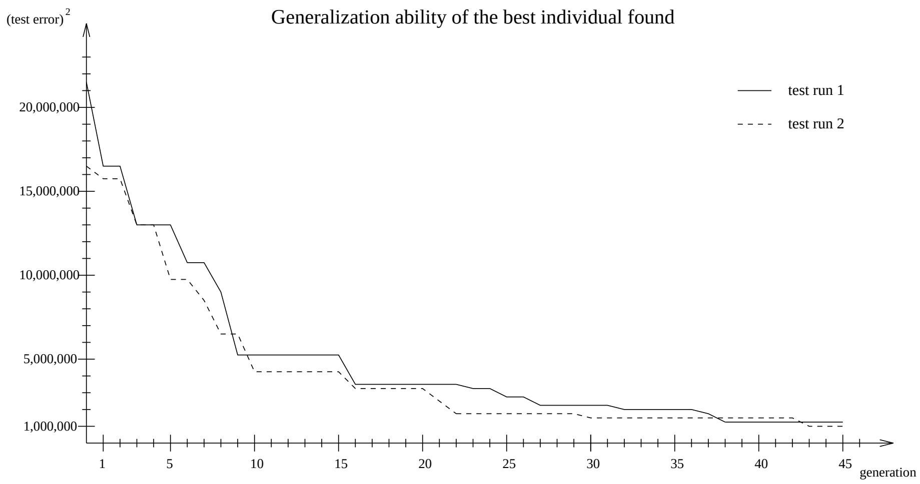

Figure 4.10: Fitness of the best individuals for generalizing the 6–bit addition.

the most common features of most solutions are one large single layered sub–network and additional multi layered sub–networks.

There is however a problem that should be regarded. Both solutions shown in figure 4.11 have far more connections (free parameters) than there were data in the training data set. This could lead to the conclusion that the networks, as they are underconstrained, are memorizing the training data set rather than learning the target function that would allow them to generalize well. However, due to the fact that the networks were trained for just an extremely short time (40 bp–epochs) it is unlikely that overlearning has occured. This thesis is also supported by the improving fitness of the networks. The test could be repeated with letting the networks learn longer and maybe then smaller structures would be developed, as found in earlier generations[61].

Finally, it should be noted that the solutions found in both runs are not necessary perfect solutions to the 6–bit addition problem. The test error (tss), although significantly smaller than in the beginning of the evolution, was still greater than 1000 and therefore it is possible that some patterns could not be fitted correctly[62].

4.3.4 Partition Test

The partition problem is similar to the parity problem, but it takes real valued inputs rather than binary. The idea is to partition a unity square in

[61] The solutions found in generation 16 (run 1) and 22 (run 2) are *not* underconstrained!

[62] If this is true, it could be a problem of the learning horizon. Training the networks for a longer time however could lead to an overlearning as some of the networks are underconstrained. A possibility to leave this dilemma is to search for good solutions in earlier generations, where the networks are smaller (not underconstrained). Those networks might be able to capture the entire problem, when trained for a longer period of time.

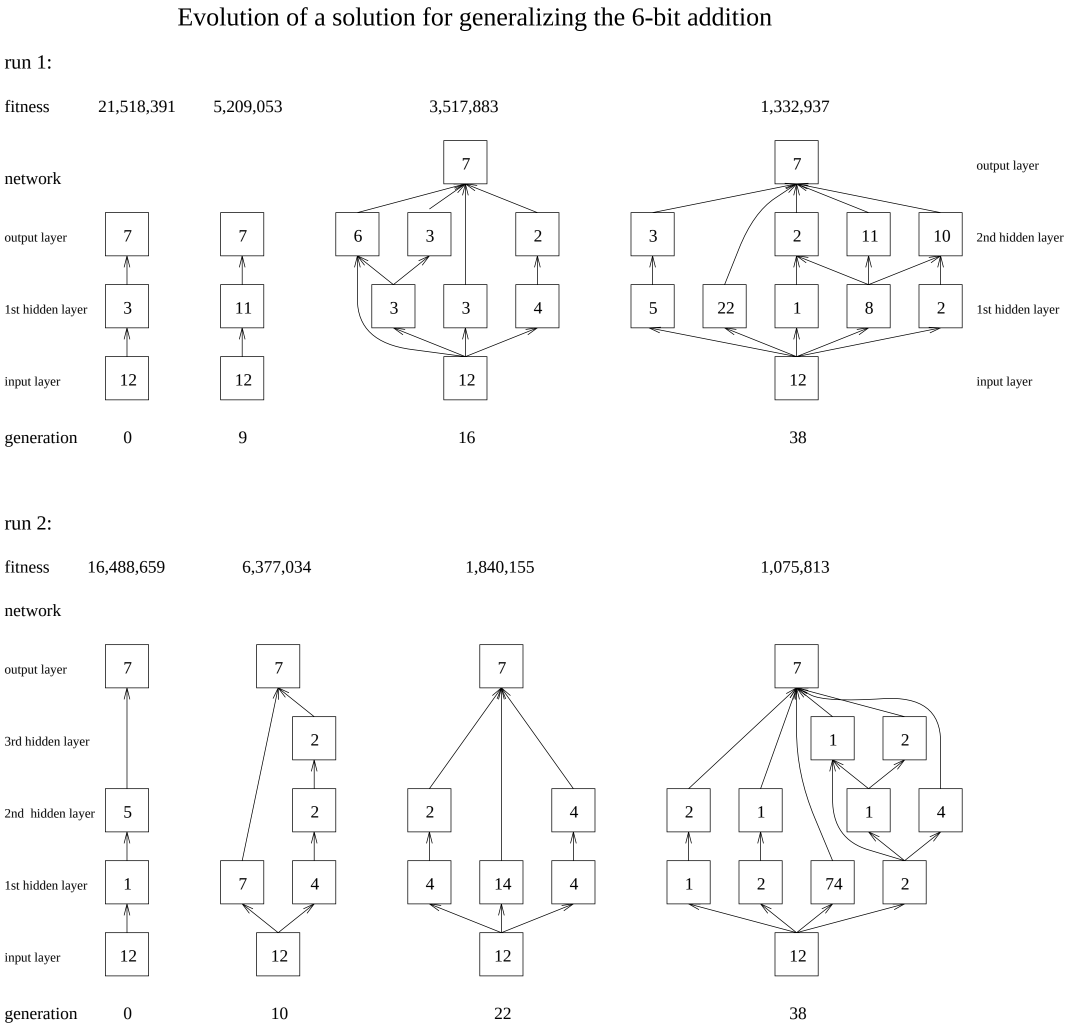

Figure 4.11: The evolution of topologies for generalizing the 6–bit addition.

the following way (see figure 4.12):

1. When x and y are both smaller than 0.5 or both greater than (or equal to) 0.5, then the output is 0.0
2. Otherwise the output is 1.0

The partition problem can be extended to higher dimensions by the following definition: "Set the output to 1.0, if there is an *odd* number of inputs that are greater or equal to 0.5. Set the output to 0.0 otherwise."

Two solutions to the two dimensional partition problem are known. Using a single layered network and extending the layer by adding units is supposed to yield improving results (with growing layer sizes). The other solution is more compact, but it requires shortcuts to the output layer, which cannot be created when using the restricted unit–cluster model.

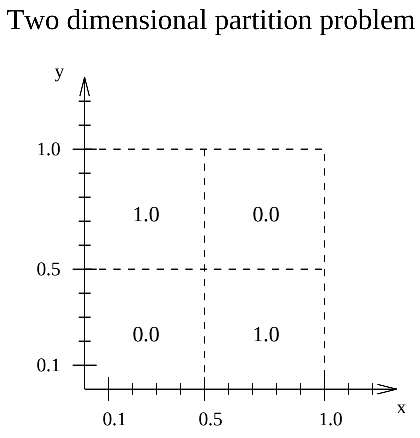

Figure 4.12: The two dimensional partition problem.

The partition problem was chosen to be three dimensional (three inputs) for this test in order to make the problem more interesting. The networks were trained with a total of 217 training data. The points chosen were: 0.0, 0.25, 0.49, 0.5, 0.75 and 1.0 combined with each other in every possible way for three dimensions. This choice contains:

1. points on the outer surfaces of the unity cube (all points with at least one coordinate set to 0.0 or 1.0)
2. points next to the borders between the different parts of the unity cube (all points with at least one coordinate set to 0.49 or 0.5)
3. points in the middle of each part of the unity cube (all points that just contained coordinate values of 0.25 and 0.75)

The test data set needed to establish the test error (generalization) contained 1331 data points. The data points were selected uniformly from the unity cube and contained all possible combinations of the x, y and z values taken from: 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 and 1.0.

Unfortunately the three dimensional partition problem turned out to be quite difficult and no solution could be found with the SCGA. Figure 4.13 shows the fitness of the best individual found during evolution when the fitness function was defined by the train error[63]. It can be seen that the fitness did not significantly improve during evolution. This is a no win situation for the genetic engine, as it has absolutely no guidance whatsoever. When there is no improvement in fitness, the genetic algorithm does not know in which direction it should adapt the population and a solution cannot be reached[64].

[63] Only one representative run has been depicted in figure 4.13, as the other one yielded similar results. Furthermore there is no sense in looking at results that were created when the fitness was defined by the test error, because if a network is not able to learn the problem at all, then it will also not be able to generalize.

[64] exception: finding a solution by accident

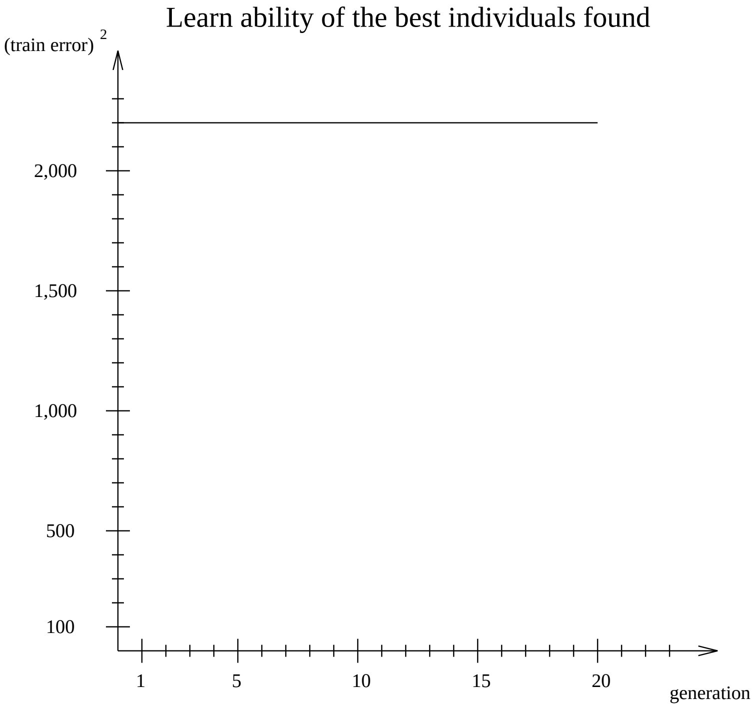

Figure 4.13: Fitness of the best individual for learning the three dimensional partition.

This however gives an opportunity to discuss different possibilities, *why* the genetic algorithm was not able to find a solution. It is interesting to reflect what reasons can lead to a non improving fitness:

- A feature of the problem, that it either will be learned perfectly, once the right topology is found, and it will not be learned at all when the topology is slightly different from the correct one. Such a feature can be observed, when regarding the double spiral problem[65]. Those problems are difficult to solve with the SCGA, as no gradual adaption of the topology is possible. The solution is either found by accident or it is not found at all.

- The evolution is strongly dependent on the starting point. Although the probability is quite low, there is still a chance that in all four test runs the initial population did not contain important individuals that would be able to learn at least a part of the problem. There are functions that are sensitive to the starting point of the evolution and a solution cannot always be found when starting with a random initial population.

[65]The double spiral problem consists of two intertwining spirals. The inputs are x, y coordinates and the network is supposed to tell on which spiral the coordinate lies.

- The learning time of the backpropagation algorithm. The networks were trained for 500 bp–epochs, but this might be insufficient for the networks to show significant performance.
- Setting the learn rate and the momentum term of the backpropagation algorithm to not favorable values can keep networks from learning a problem.
- The initial individuals were too small. When the initial individuals are too small no solution may be found, as the SCGA limits the growth of the individuals. If there is no immediate payoff for the SCGA in growing individuals, they can grow only slowly. Therefore it can take many generations till they are large enough to be able to encode a large network that might solve the problem.
- Shortcuts to the output layer are a necessity to solve the problem. If this is the case, then it is impossible to find solutions using the restricted unit–cluster model, as this feature has not been incorporated in the alphabet.

It can be seen that there are several possibilities that can lead to a failure in finding a solution with the SCGA. Most of the possible reasons can be eliminated by adjusting parameters. This however is not always a trivial task. Adaptive learn algorithms[66] [30] and longer test runs[67] can help, but do not ensure that a solution will be found.

4.3.5 Breast Cancer Prediction Test

The breast cancer prediction is a real world problem. The idea is to predict from 9 input attributes, whether the patient has breast cancer or not. The output is therefore binary, but it has been coded with two output units, where each output unit represents one state. The real world data has been taken from a standard database called "Proben 1" [23]. This database contains several data sets for evaluating performances of neural networks.

Two separate tests were conducted. The first test was supposed to evolve networks that would be able to learn a larger data set of 699 data samples (fitness defined by the train error). For the second test two different data sets were used. The networks were trained on a dataset with 350 samples and then their generalization ability (fitness defined by the test error) was tested on a test set that contained 175 samples.

In order to evolve networks that would be capable of learning the cancer prediction two test runs were conducted. Their results are depicted in figure 4.14. The two independent runs were designed differently and this explains

[66] e.g. a backpropagation algorithm with adaptive learn rate and momentum

[67] more and larger individuals, that are trained for a longer time

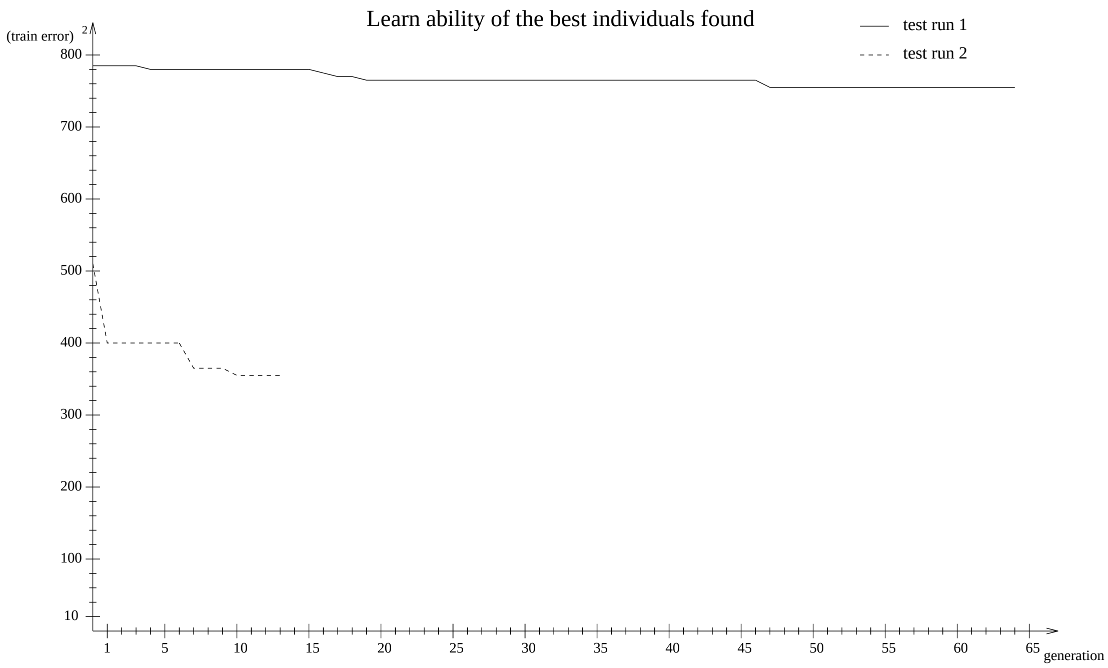

Figure 4.14: Fitness of the best individuals for learning the breast cancer prediction.

the large difference in the outcome. In the first run the amount of backpropagation epochs was set to small value (40) and the amount of generations was set to a large value (80). This was supposed to test the behavior of the genetic system when it has many generations to evolve a solution, but only a rough fitness estimation. The second run was designed in the opposite way, giving the genetic system a more detailed fitness information, by letting the networks learn for 150 backpropagation epochs, but limiting the amount of generations to 13. Both tests showed rather disappointing results, as the networks did not improve their fitness significantly during evolution. This however holds only when regarding figure 4.14. Interesting results were obtained as will be shown later, when a close up will be taken on the individuals.

The results of the generalization test run are depicted in figure 4.15. The networks were trained for 150 backpropagation epochs and then they were tested using the test data set in order to calculate their fitness. The first test run shows a similar behavior as already observed in the tests for the learning ability. The second test run was designed to let the genetic algorithm optimize the networks for many generations more (80). An interesting outcome can be observed in generation 70. The fitness of the best individual jumps from 34 to 1. This is a quite unusual behavior for the genetic algorithm, especially as it seemed to have already converged on a solution.

In order to explain this unusual behavior a close up on the evolution had to be taken and is presented in figure 4.16. This close up revealed much

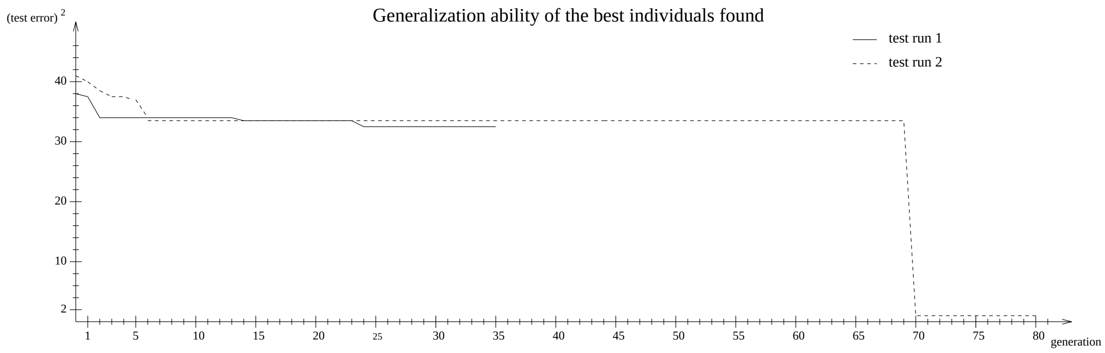

Figure 4.15: Fitness of the best individuals for generalizing the breast cancer prediction.

additional information about the solutions found by the SCGA. Three networks are shown in figure 4.16. As the best network found in generation 6 was lost during table adaption networks of similar quality were elected to be presented. Those are the best networks found in generation 7 and 11. All networks depicted were trained several times with the backpropagation algorithm for a longer period of time (400 epochs) and their train and test errors were recorded each 50 epochs starting with epoch 0. A representative interpolation of the learn and generalization progress for the three networks is also depicted in figure 4.16.

The first insight that was given by this close up revealed the fact that all networks learned and generalized the data quite well. It could be observed that most of the data was learned and generalized except some outliers that could not be integrated.
Furthermore, the best network found in generation 70 did not reach the fitness of 1.0 during several independent training runs. This leads to the conclusion that the initial weights of this network must have been extremely favorable during the evolutionary run to allow this network to have such a good fitness. This suspicion is also supported by the fact that the best quality of generation 71 was 38.6. If the best network found in generation 70 had been able to generalize that well again, it easily would have protected its translation table entries and would have appeared with a similar fitness in generation 71.
Finally, it can be observed that the network found in generation 70 really performs slightly better than the two networks found in generation 7 and 11. When looking at the generalization error depicted in figure 4.16 one can see that this network has a slightly better generalization performance than its competitors. When looked at in the close up it can also be seen that this network has a better learning performance after training it for 400 epochs and that it also is less sensitive to overlearning[68]. Especially the

[68] Overlearning happens later; after more backpropagation epochs than when using the other networks.

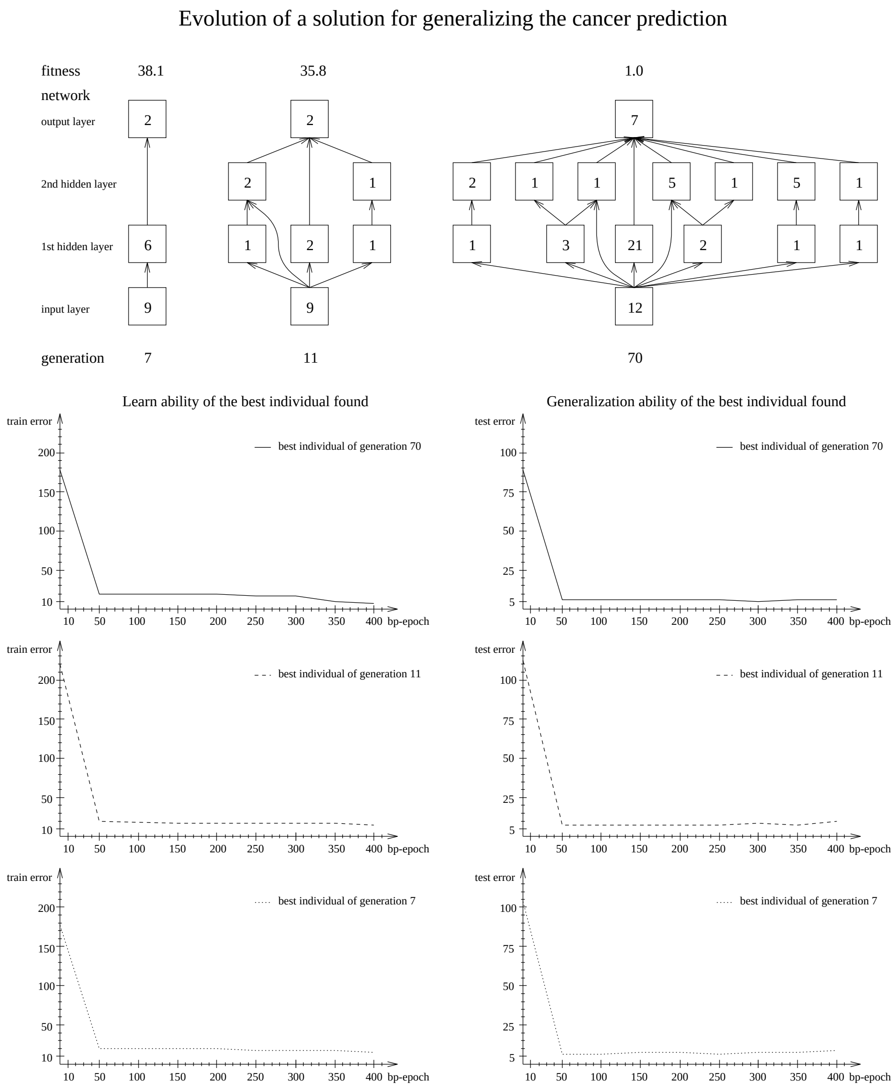

Figure 4.16: Evolution of solutions for the breast cancer prediction problem.

minor sensitivity to overlearning is an interesting result as one would expect a larger network to be much more sensitive to overlearning.

In conclusion it must be said the the breast cancer prediction problem:

1. is too simple for employing the SCGA to find a solution, as even simple and non–modular networks constitute good solutions.

2. was not solved completely, because there always remained single data points that could not be generalized by the solutions found with the SCGA.

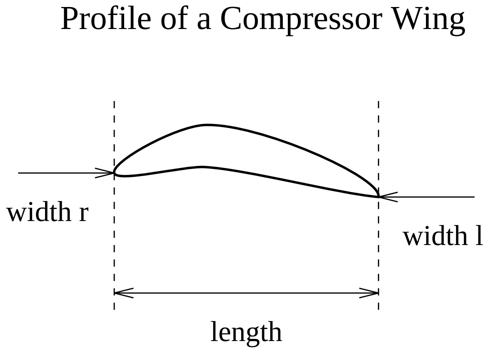

Figure 4.17: Profile of a compressor wing with some stability relevant parameters.

4.3.6 Profile Construction Test

The profile construction is also a real world application. The compressor part of a plane engine contains many twisted wings of different sizes. In order to ensure maximal performance of the engine those wings have to have a certain individual size and shape. One step that has to be done during the construction of a turbine is to find an optimal shape for those twisted wings. This is accomplished by constructing the profiles of those wings[69] (see figure 4.17). The profiles must however comply to certain stability criteria and size limitations. A few of those criteria and limitations are depicted in figure 4.17. The data set contains 11 input criteria that define the constrains[70]. The output are then 11 spline parameters that define the shape of the profile.

The design of those profiles is usually done manually[71], but since there is a small data base of working profiles an attempt can be made to find a neural network that will automate the construction. Actually as the data set was extremely small (63 samples) the only reasonable attempt was to find networks that will learn the data. Although this is not the main goal in automating the construction process, as generalization is needed, it can give some insight to how difficult it is to just learn the data.

The learn performance of the best networks is depicted in figure 4.18. The networks were trained for 400 backpropagation epochs and it took only a few generations (ca. 10) to find good solutions. The solutions found had many different structures (also multi–layered), but there is not much sense in showing them all as it is obvious that larger networks are able to learn the data better than smaller ones. There is however the possibility to concentrate on the smallest networks that performed well and were found by the SCGA. Those networks were compared to a network that has been found manually in an earlier study. The result is depicted in figure 4.19. The manually adapted network had a single hidden layer with 20 hidden units. The SCGA, on the

[69] A profile is one (out of many) cross–sections of such a wing.
[70] The data has been normalized due to security reasons.
[71] There are also few evolutionary approaches.

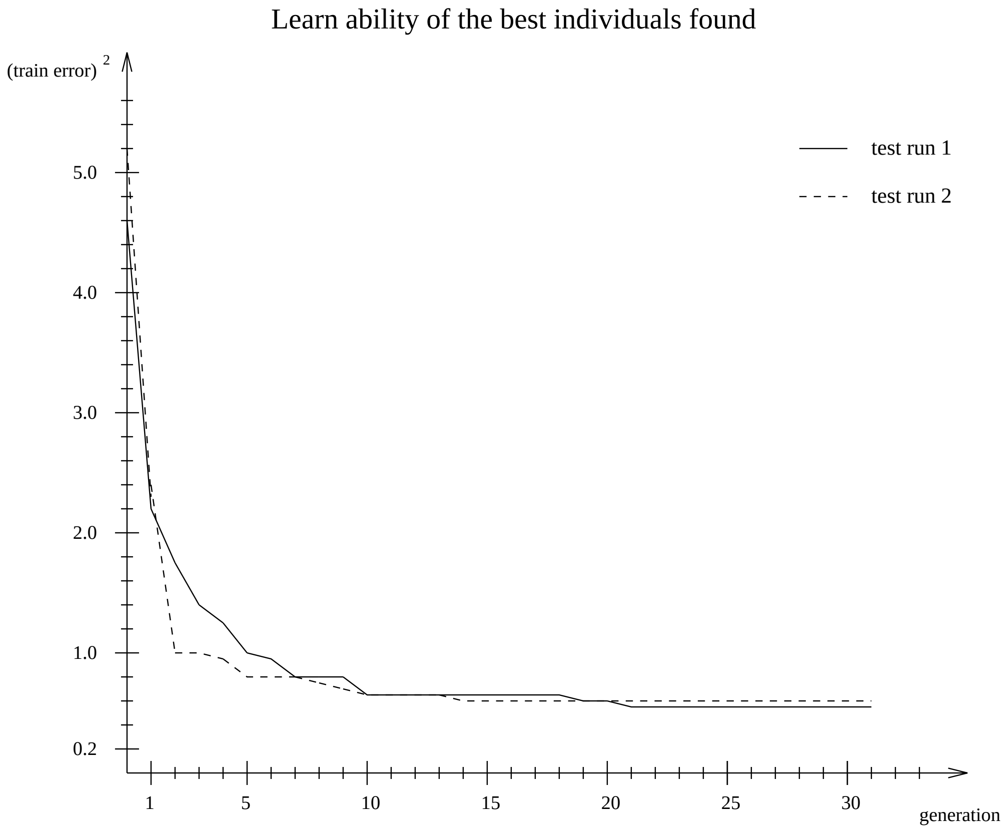

Figure 4.18: Learn ability of the best individuals for the profile construction.

other hand, recommended networks with one hidden layer and at least 26 hidden units. Those networks were able to learn the data faster and better.

Nevertheless, it should be noted that both solutions were heavily underconstrained, due to the small data set. Therefore the only conclusion that can be made is that the SCGA is able to find networks that will learn the training data. It is likely that a larger data set will be learned as well, but it is impossible to make a definite prediction.

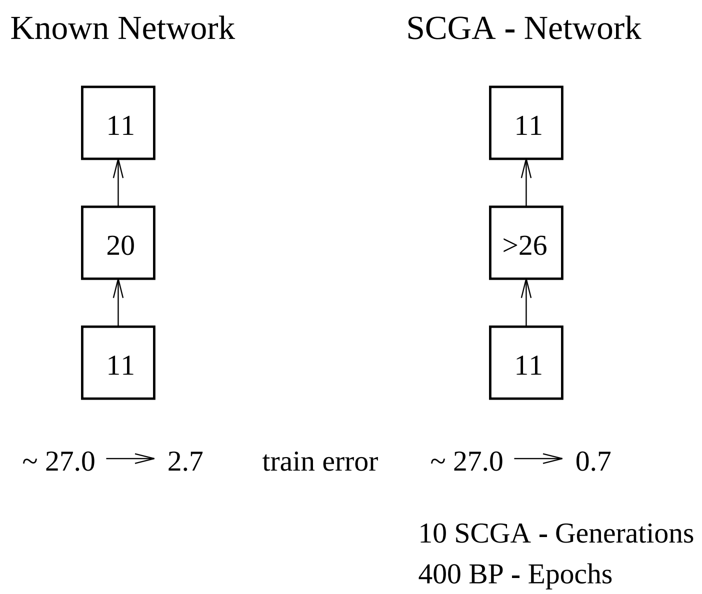

Figure 4.19: Solution found by manual adaption vs. solution found by the SCGA.

4.4 Conclusion

In this chapter the evolution of topologies of feedforward neural networks with the semantic changing genetic algorithm has been described.

A review of different approaches to the design of a genetic coding for the optimization of neural networks has been presented first. Two main categories of genetic codings were mentioned:

1. direct encoding scheme
2. indirect encoding scheme

In a short introduction to the approach elaborated in this study, its main motivation was described. The idea was to construct a genetic coding that would allow to encode modular structures of neural networks in order to fit the optimization strategy of the semantic changing genetic algorithm.

This lead to the establishment of the unit–cluster model. In the unit–cluster model mainly units are encoded in the genotype, while the connectivity is derived from the decoding process. It constitutes a way of encoding

1. clusters
2. modules
3. sub–networks

close to each other on the genotype.

In order to analyse the advantages and disadvantages of using the unit–cluster model for the definition of a genetic coding an investigation was made how well the quality criteria established in section 3.2 have been met:

- completeness
- closure
- proximity
- short schemata
- compactness
- non–isomorphism
- modularity

Furthermore some additional advantages of the unit–cluster model were mentioned:

- small alphabet

- linear growth of genotypes
- encoding of relative unit positions

The analysis yielded that most of the criteria were met or violated in just a minor way, when the complete system – the higher and the lower level coding – was regarded. One highly important criterion however was explicitly not met – the completeness criterion.

In order to meet the completeness criterion an extension of the unit–cluster model was proposed. The extension enabled the encoding of additional connections on the genotype, next to the backbone structure that was derived from the decoding process.

Finally, the restricted unit–cluster model was introduced. This model allows the definition of static input and output layers. It is supposed to speed up the evolution as it provides a way to incorporate a priori knowledge, which is often given through the problem data sets. The restricted unit–cluster model however has a problem meeting the completeness criterion. Two kinds of topologies cannot be encoded:

1. topologies without a monolithic input and output layer[72]
2. topologies with shortcuts to the output layer

Some suggestions have been made on how to get rid of those limitations, but the unchanged restricted unit–cluster model has been used in all tests.

The importance of an appropriate fitness function has been explained and the selection of the backpropagation algorithm for obtaining fitness measures has been reasoned. Several single fitness measures have been discussed:

- amount of units
- amount of links
- train error
- test error

Furthermore some examples of combined fitness measures were presented and the following test schedule for applying different fitness functions has been suggested:

1. train error
2. combination of train error, amount of units and amount of connections
3. test error or combination of test error and train error

[72]Both layers are treated as one cluster each.

Four tests were conducted in order to test the performance of the genetic system.

1. The 6–bit addition test showed that modular feedforward networks, which have the ability to learn and generalize, can be evolved.

2. The partition problem gave an opportunity to discuss some possibilities why sometimes no solution can be found.

3. The breast cancer prediction problem represented a class of problems, which are simple enough so that although they can be solved by the genetic system there is no need for applying it.

4. The profile construction test revealed that the genetic system is able to find simple better solutions than those found manually and that it is sometimes fruitful to search for solutions in early generations, before the evolution has converged completely.

The tests gave some insight to how the genetic system made out of the semantic changing genetic algorithm, the restricted unit–cluster model and the backpropagation algorithm works, for which problems it is worthwhile to use it and where its limitations are.

Chapter 5

Conclusion

This chapter is designed to present two aspects. First, the achievements of this work are reviewed and then an outlook is given to what further improvements and work have to be done to establish the system presented in this study. The major two achievements are the genetic changing genetic algorithm and the unit–cluster model. The main future tasks are concerned with a refinement and extension of the presented approach and with some suggestions to tests that would clearly exhibit the capabilities and limits of the developed system.

5.1 Achievements

Chapter 1 gives a short introduction to the purpose of this study.

The importance of the topology design for neural networks is mentioned and it is argued that feedforward neural networks are only able to solve a problem, when an appropriate topology is chosen.

Furthermore, the application of genetic algorithms for the topological optimization of neural networks is justified by stating that this approach is more flexible to the design of different topologies than heuristic approaches and that it is computationally less expensive than "brute force" methods.

Finally, the goals of this work are defined. The main goal is to construct a genetic system that is able to find problem dependent modular topologies for feedforward networks that can be trained with a discrete learn algorithm to learn and generalize a problem.

Chapter 2 constitutes a short introduction to neural networks. It describes the biological motivation and its simple artificial model, explains what feedforward networks are, mentions the advantages of modular feedforward networks, presents a learn algorithm for feedforward networks and discusses some important parameters of feedforward networks that have to be set carefully for the networks to solve a problem.

The motivation for creating artificial neurons is founded in the differences between the abilities of common computers and the human brain.

The functionality of biological neurons is described. The level of abstraction chosen to create a model of a neuron distinguishes between the following parts of a neuron:

- synapses
- dendrites
- cell body
- nucleus
- axon

The established parts of a biological neuron are modeled in an artificial neuron as follows:

- connection weights (synapses)
- connections (dendrites)
- sum function (cell body)
- activation function and bias (nucleus)
- output (axon)

The focus is then directed to a special class of artificial neural networks that can be build with those artificial neurons – feedforward networks. A definition is given that states that feedforward networks are just allowed to propagate input signals unidirectional. Their layered structure, which includes input, hidden and output layers is explained.

The class of feedforward networks is divided into modular and non–modular feedforward networks. A comparison of those two classes is made and the following advantages of modular feedforward networks are elaborated:

- lower complexity
- more suitable for interference problems
- limitation of equal solutions

A learn algorithm – the backpropagation algorithm – is shortly presented. Its main idea and mathematical model are described. The backpropagation algorithm is classified to be a gradient descent search algorithm that is able to quickly find a local solution by iteratively adapting the weights of a feedforward network.

Finally, the influence of several parameters on the learning and generalization capabilities of networks is described. Those parameters are:

- topology
- weight initialization
- learn parameters
- train and test patterns

The analysis shows that there are many parameters that have to be set in an appropriate and problem dependent way for networks to be able to solve problems.

Chapter 3 covers the field of genetic algorithms. The importance of the genetic coding is emphasized, two forms of the simple genetic algorithm are presented and finally the semantic changing genetic algorithm that has the ability to dynamically adapt the genetic coding through population analysis and one derivation of it are developed. Finally, the capabilities of the different approaches are compared to each other through tests.

The importance of the genetic coding is discussed first and the following criteria are established to measure the quality of a genetic coding:

- completeness
- closure
- proximity
- short schemata
- compactness
- non–isomorphism
- modularity

An outline of the simple genetic algorithm (SGA) as introduced by Holland [9, 10] is presented, the function of the genetic operators – selection, crossover, mutation and reduction – that are used in this research is described and the extended simple genetic algorithm (ESGA) is derived from the simple genetic algorithm. The extended simple genetic algorithm can operate on genotypes of variable length.

The need for an extension to the genetic algorithm is elaborated next. Several features are presented that are desirable, but not covered by neither the SGA nor the ESGA. The features are:

- controlled growth of the genetic code
- automatic module definition
- extended module distribution

- module isolation
- automatically meeting the proximity criterion
- supporting closure
- integrating knowledge

The semantic changing genetic algorithm (SCGA) is introduced. It includes most of the desired features. The SCGA introduces an add–on extension to the simple genetic algorithm, which dynamically modifies the coding of the genotypes using population analysis and Darwinian evolution. The next step is then to introduce the extended semantic changing algorithm (ESCGA) that uses the extended simple genetic algorithm as its basis. The ESCGA is analyzed critically and the weak points of the extension – lost growth control and interfering operations – that finally lead to its dismissal are pointed out.

Finally, three tests are conducted to compare the capabilities of the three approaches – ESGA, SCGA and ESCGA. The tests show that the newly introduced algorithms are capable of solving genetic problems. Furthermore, they show that there is a class of functions for which the SCGA clearly outperforms the ESGA. The ESCGA, on the other hand, turns out to be a poor compromise between the ESGA and the SCGA. It lacks the dynamics of the ESGA and it lacks the control possibilities the user has using the SCGA.

The tests indicate that the SCGA might be more useful for solving some problems than the ESGA and that this strongly depends on the modularity of the problem, as well as on the possibility to use simple modules to construct larger ones consecutively[1].

Chapter 4 discusses the evolution of topologies of feedforward neural networks with the semantic changing genetic algorithm. Different categories of genetic codings of neural networks are presented and the unit–cluster model, which supports the coding of modular feedforward networks with a "backbone" structure, is developed. The unit–cluster model is analyzed critically and two extensions to the model are elaborated – the extended unit–cluster model and the restricted unit–cluster model. Finally, single and combined fitness measures for measuring the quality of topologies with regard to a given problem are presented and several tests are conducted in order to investigate the capabilities of the genetic system composed of the semantic changing genetic algorithm and the restricted unit–cluster model.

A review of different approaches to the design of a genetic coding for the optimization of neural networks is presented first. Two main categories of genetic codings are mentioned:

1. direct encoding scheme

[1] e.g. the way fractals are designed

2. indirect encoding scheme

In a short introduction to the approach elaborated in this study, its main motivation is described. The idea is to construct a genetic coding that will allow to encode modular structures of neural networks in order to fit the optimization strategy of the semantic changing genetic algorithm.

This leads to the establishment of the unit–cluster model. In the unit–cluster model mainly units are encoded in the genotype, while the connectivity is derived from the decoding process. It constitutes a way of encoding

1. clusters
2. modules
3. sub–networks

close to each other on the genotype.

In order to analyse the advantages and disadvantages of using the unit–cluster model for the definition of a genetic coding an investigation is made how well the quality criteria established in section 3.2 are met:

- completeness
- closure
- proximity
- short schemata
- compactness
- non–isomorphism
- modularity

Furthermore some additional advantages of the unit–cluster model are mentioned:

- small alphabet
- linear growth of genotypes
- encoding of relative unit positions

The analysis yields that most of the criteria are met or violated in just a minor way, when the complete system – the higher and the lower level coding – is regarded. One highly important criterion however is explicitly not met – the completeness criterion.

In order to meet the completeness criterion an extension of the unit–cluster model is proposed. The extension enables the encoding of additional connections on the genotype, next to the "backbone" structure that is derived from the decoding process.

Finally, the restricted unit–cluster model is introduced. This model allows the definition of static input and output layers. It is supposed to speed up the evolution as it provides a way to incorporate a priori knowledge, which is often given through the problem data sets. The restricted unit–cluster model however has a problem meeting the completeness criterion. Two kinds of topologies cannot be encoded:

1. topologies without a monolithic input and output layer[2]
2. topologies with shortcuts to the output layer

Some suggestions are made on how to get rid of those limitations, but the unchanged restricted unit–cluster model is used in all tests.

The importance of an appropriate fitness function is explained and the selection of the backpropagation algorithm for obtaining fitness measures is reasoned. Several single fitness measures are discussed:

- amount of units
- amount of links
- train error
- test error

Furthermore some examples of combined fitness measures are presented and the following test schedule for applying different fitness functions is suggested:

1. train error
2. combination of train error, amount of units and amount of connections
3. test error or combination of test error and train error

Four tests are conducted in order to test the performance of the genetic system.

1. The 6–bit addition test shows that modular feedforward networks, which have the ability to learn and generalize, can be evolved.
2. The partition problem gives an opportunity to discuss some possibilities why sometimes no solution can be found.

[2]Both layers are treated as one cluster each.

3. The breast cancer prediction problem represents a class of problems, which are simple enough so that although they can be solved by the genetic system there is no need for applying it.

4. The profile construction test reveals that the genetic system is able to find simple better solutions than those found manually and that it is sometimes fruitful to search for solutions in early generations, before the evolution has converged completely.

The tests give some insight to how the genetic system made out of the semantic changing genetic algorithm, the restricted unit–cluster model and the backpropagation algorithm works, for which problems it is worthwhile to use it and where its limitations are.

5.2 Future Work

This section contains an incomplete list of suggested improvements, changes and tasks that can be carried out to strengthen and elaborate the usefulness of the approach presented in this study.

5.2.1 Improvements and Changes

There are possible changes that could result in an improved performance of the genetic system. Some of them are mentioned here:

- **full modularity and completeness**
  The restricted unit–cluster model has some limitations to topology encoding – monolithic input and output layers and no possibility to encode shortcuts to the output layer. This however are quite serious restrictions and extensions[3] should be found to eliminate them.

- **topologies without hidden units**
  The genetic system is not designed to cope with networks that have no hidden units. Such a network can be encoded, but the system does not connect the input layer to the output layer. This was not crucial for this study as only problems that need hidden units could exhibit the modularity features of the restricted unit–cluster model, but it should be changed as sometimes good solutions without a hidden layer exist and a hidden layer can make the problem artificially harder [1].

- **genetic adjustment of learn parameters**
  A learn algorithm like backpropagation usually contains different crucial parameters that have to be defined by the user. It is however not always trivial to find good values for those parameters as can be seen

[3] see suggestions in section 4.1.6

in the example of learn rate and momentum used by the backpropagation algorithm. This can be solved by the genetic algorithm when the parameters are encoded on the genotype.

- **limitation of crossover points**
  The crossover operator applied to the lower level genetic coding could be limited to points that lie between encoded table entries[4]. This would make the crossover operator act less disruptive. Without this limitation (the way it is handled now) the crossover operator has a huge mutational side effect, it can make both offsprings address up to two table entries that are not being addressed by their parents.

- **different mutation**
  The mutation operator applied to the lower level genetic coding has a similar effect. Changing one bit in the lower level genetic coding may result in a highly different higher level genetic coding. In order to prevent this from happening a similarity of table entries could be calculated and the mutation could be applied to change the lower level genetic coding from addressing one table entry to the next similar table entry.

- **different selection of crossover points**
  After the limitation of crossover points they also could be selected differently. Instead of choosing same positions in both parents (lower level coding) just the length of the swapped parts would have to be the same. This would allow modules to migrate and thus the encoded substructures could be used at different places in the network.
  Another advantage of a restricted crossover point selection is that it enables the use of higher level codings that do not automatically comply to the closure criterion. One would have to define some order for the letters that would state which letters can follow after which letters in the higher level coding to ensure a valid coding and could then limit crossover points to positions that would comply to those definitions. With an increasing amount of letters, a highly restrictive letter hierarchy (grammar) and a short lower level genetic coding this could however lead to situations where two parents would not be able to produce offsprings[5].

- **different importance**
  The importance of a translation table entry is dependent on how often the table entry is being addressed by the current population and on the fitness of the individuals that access the table entry. This however does not need to be the best measure. The experiments showed that an

[4]With a six bit encoding of a table entry these are points after every six bits starting with (before) the first bit.

[5]This would happen, if no sub–string of equal size could be found in both parents that would have a compatible beginning and end letter when translated into the higher level genetic coding

outstanding solution may be lost during table adaption and that the fitness function has to be designed to overvalue better individuals (tss^2 instead of the simple tss) in order to protect their table entries from being overwritten. This could be solved by e.g. defining the importance of a translation table entry through the fitness of the best individual that accesses the table entry. The table entries addressed by the best individuals could then be easily protected[6].

- **restricted selection of modules**
  A restricted selection of donor sub–strings that enter the translation table during table adaption could lead to a more compact coding. Useless modules could be refused or the modules could be stripped of useless sub–strings. A useless sub–string is e.g. ><>< as it does not take any direct influence on the topology encoded[7].

- **different placement of modules**
  Modules (substrings) enter the translation table at randomly selected places. It could be an advantage to apply a more sophisticated method and individually select a table space for each module. One could e.g. establish similarities between the modules that are currently in the table and are supposed to be overwritten and the modules that are supposed to enter the translation table. It would then be possible to replace table entries with modules that are similar and not completely different. This would make the table adaption be less disruptive to the population.

These are just a few suggestions of how the existing system may be improved and changed. There are probably many more, but those presented here constitute a good starting point.

5.2.2 Different Tests

The presented approach to the genetic optimization of topologies should be tested further than it could be done in this study in order to obtain an overview over problems that can be solved with this system and over the quality of those solutions. This section presents some test setups that would give a more detailed insight to the performance of the presented and maybe already improved approach:

- **define problem classes**
  The first step should be to define distinct problem classes[8]. Different examples out of those distinct problem classes could then be selected and tried to be solved by the genetic system. This would give insight

[6] This is only possible, if the individuals do not access all table entries at once – if the translation table is large enough, but this is desired anyway.

[7] It may take influence by marking off clusters, but this is *not* a desired side feature.

[8] The classification of problems may vary with respect to the selected learn algorithm.

to which problem classes can be solved, which problem classes are too difficult and which problem classes are too simple for the system to be applied.

- **different parameter settings**
  Another quite straight forward approach is to run the genetic system with different parameter settings for each problem. The most crucial parameters to be investigated are:

  - parameters of the learn algorithm (here: learn rate and momentum)
  - length of evolution (amount of generations)
  - population size
  - length of the lower level genetic individuals (size of the higher level coded initial individuals)
  - elitist vs. non–elitist strategy
  - size of translation table
  - amount of table spaces updated each generation
  - mutation rate for modules entering the translation table
  - enforcement parameter for the preferred length of modules entering the translation table

  Tests with different parameter settings would reveal how robust the system is to the setting of those parameters and which settings are needed for which kinds of problems.

- **extended vs. restricted unit–cluster model**
  Some tests could also be conducted to show the performance of the semantic changing genetic algorithm applied to the extended unit–cluster model in comparison to its performance with the restricted unit–cluster model. The extended unit–cluster model has the advantages that it already complies to the completeness criterion and that it has the possibility to develop topologies with fewer input units, which might be of advantage when the input data contains redundant parameters.

- **independent tests**
  Another test setup could give some insight to the performance of the semantic changing genetic algorithm and the extended or restricted unit–cluster model independently. A first attempt to just measure the performance of the semantic changing genetic algorithm has already been presented in section 3.9. The restricted unit–cluster model could be tested in connection with the extended simple genetic algorithm. Independent tests of those two newly developed parts could aid gaining more understanding of their behavior.

- **SCGA vs. ESGA**
  A crucial test would be to compare the performance of the semantic changing genetic algorithm to the extended simple genetic algorithm when both are applied to the extended or restricted unit–cluster model. The test would show, whether the semantic changing genetic algorithm really outperforms the extended simple genetic algorithm. If the result turned out positive, than there would be clear evidence that the semantic changing genetic algorithm is a superior approach to topology construction of modular feedforward networks with respect to the unit–cluster models and justify its practical application.

- **different learn algorithms**
  It would also be interesting to investigate the behavior of the genetic system with different learn algorithms. As the topologies are adapted with respect to the chosen learn algorithm (here: backpropagation) same topologies may be of different quality depending on the learn algorithm used to evaluate their fitness[9]. Those tests could reveal which learn algorithm is the one that enables the genetic system to find more solutions quicker and more reliable[10].

More test setups could be found, but those presented here would reveal many abilities and limits of the genetic system and would aid to the decision under which circumstances and for which problems the presented and possibly improved system should be used.

5.3 Concluding Remarks

In conclusion it can be said that the developed approach made out of the semantic changing genetic algorithm and the restricted unit–cluster model is a promising attempt to the construction of feedforward neural networks. The approach includes novel strategies to network design that are supposed to be a step in the direction of enlarging the capabilities of systems made for automated neural network design.

Nevertheless it should be noted that there is still much work to do in order to prove and establish the superiority of those strategies.

[9] Some networks that cannot be trained to learn the problem using backpropagation might be able to capture it when trained with an evolutionary approach.

[10] There can be a tradeoff between algorithms that find more solutions and algorithms that find solutions faster. An evolutionary learn algorithm might be able to train more different networks and thus find more solutions than backpropagation, but it also can take a long time as each evaluation of a network is more time consuming.

Appendix A

Software Manual

THE USERS GUIDE TO THE NEURAL NETWORK GENERATOR

by

Rafał Sałustowicz
March 1995

TABLE OF CONTENTS

0. ABOUT THIS GUIDE

This is a quick guide that describes how to install and use the neural network generator. For a description of how the algorithm works You should consult the diploma work "A Genetic Algorithm to Construct Topologies of Neural Networks", Rafał Sałustowicz, Technical University of Berlin, 1995. Furthermore it should be noticed that the license for the backpropagation algorithm is taken from the book "Explorations in Parallel Distributed Processing", MIT Press. The license text can be found in NETWORK_GENERATOR/BP_FILES/SRC/license.txt. The license for the original GENESIS package which is a part of the genetic engine is a standard GNU license that comes with any GNU product. Both licenses state that the products are free for research use and that they may be modified. The network generator package has been tested carefully for Silicon Graphics Challenge and Sun workstations. This particular version is adjusted for the Silicon Graphics Challenge, but should run fine on Sun workstations if recompiled (some minor changes might be required). Nevertheless the package is provided with no warranty.

1. GETTING STARTED
If You have the compressed archive file network_generator.tar.Z, place it in a directory that is convenient for You (suggestion: Your home directory). Then type:

uncompress network_generator.tar.Z

it will yield You the file

network_generator.tar

then type:

tar xf network_generator

this will extract all the files including the following directory structure (You could now delete the network_generator.tar file, but it might be advantageous to compress and keep as a backup). Before looking at the directory structure You should first type:

cd NETWORK_GENERATOR

and then

INSTALL_ALL

You do not need to run INSTALL_ALL, if all the binaries (ga, bp, best_individuals, individual_to_bp_matrix) and required subdirectories (NEURAL_TABLES, LAST_GENERATIONS) exist. If, however, You are unsure or have copied it from a different computer system, You should run INSTALL_ALL in order to make sure that all the binaries are updated. Notice: an installation on a different computer system should be made "by hand" using the Makefiles, because the INSTALL_* shell scripts suppress all error messages.

The directory structure after a successful installation looks then as follows:

- **NETWORK_GENERATOR/**
  This is the main directory of the network generator where You should start the experiments. It contains all relevant subdirectories, the main files to setup a genetic session and parameter input files to run it.
  Files:
  - **INSTALL_ALL**
    this is the command You should type first, if you do not have all the executables installed yet.

- **setup**
  this is the setup file of the GENESIS package, it can be used to generate a new in.* file. Read the NETWORK_GENERATOR/DOC/genesis.txt for detailed information. (Suggestion: It is often easier to edit the in.NN file than to generate a new one)
- **report**
  a tool from the GENESIS package for making a report; read genesis.txt
- **ga.script**
  runs the generator in a "silent"/ background mode with no visual output. Creates the host.log file to show You on which host the genetic engine is running, so You do remember it even a few days later.
- **ga**
  the genetic engine; creates the result files:
  * **out.***
    contains the fitnesses of each generations; read genesis.txt to understand the format.
  * **dump.***
    files containing the population of each generation (lower level coding)

  Furthermore all translation tables are saved. The default names of the files are
  NEURAL_TABLES/neural.table.NN.<experiment>.<generation>
  The saving schedule and the filenames can be modified in the neural.param.NN file.

  All parameter files for the genetic engine have the *.NN format!
- **in.NN**
  an example of an in.* file, contains input parameters for the GENESIS part of the genetic engine; explained later
- **neural.param.NN**
  contains input parameters for the network construction part of the genetic engine; explained later
- **alphabet.NN**
  the alphabet of the higher level coding; explained later

• **NETWORK_GENERATOR/BP_FILES**
contains the backpropagation simulator bp, examples of input files and an example pattern file. Examples of input files are: bp.net - a network_file, bp.str - a script file, bp.tem - an output mask. The bp.script runs the simulator in a "silent"/ background mode.

• **NETWORK_GENERATOR/BP_FILES/SRC**
source files for the bp, a Makefile and the INSTALL_BP file.

- **NETWORK_GENERATOR/DOC**
  contains all the documentation available on-line. Additional information for the backpropagation algorithm can be found in the book "Explorations in parallel distributed processing" by D.E. Rumelhart, MIT Press, 1988.

- **NETWORK_GENERATOR/LAST_GENERATIONS**
  directory where You can save the result generations.

- **NETWORK_GENERATOR/NEURAL_TABLES**
  directory where the translation tables for the lower genetic representation are automatically saved.

- **NETWORK_GENERATOR/SRC**
  contains all the source files for the genetic engine and an INSTALL_GA file that allows You to install the genetic engine.

- **NETWORK_GENERATOR/TOOLS**
  contains two tools:

  - **best_individuals**
    transforms individuals from lower to higher level coding.

  - **individuals_to_bp_matrix**
    transform individuals from (higher level coding) to backpropagation *.net files.

- **NETWORK_GENERATOR/TOOLS/SRC**
  sources and Makefile for the tools

- **NETWORK_GENERATOR/TOOLS/NETWORK_TO_PS**
  network to postscript converter
  !!! NOT ADAPTED FOR SGI CHALLENGE !!!
  To get it to run You have to compile individuals_to_matrix, this is a tool which translates the higher level coding to the input format of the converter. Then You have to use the Makefile to make net2ps - the converter.
  NOTICE: source code adaption necessary!

2. THE GENETIC ENGINE

The genetic engine consists of three parts - the GENESIS package that implements the simple genetic algorithm and an user interface, the semantic changing algorithm part that implements the table adaption and the backpropagation program. The backpropagation program is a stand alone program and discussed in the following section. The network generator works roughly like that: The two genetic parts of the package work closely together. Each generation POPULATION_SIZE to 2·POPULATION_SIZE individuals are evaluated, depending on how many individuals are affected by the table adaption. Each individual is evaluated using the bp-algorithm.
To change the fitness function one has to edit the SRC/neural_fitness.c and/or BP_FILES/bp.str and/or BP_FILES/bp.tem files. To use a neural.param file with a different name the definition in SRC/neural.h must be changed. To use a different evaluation program than the bp You might have to change the translation of the higher level coding to the network input files in the file SRC/neural_bp_link.c.

How to run the network generator:
There are three data input files needed to run the generator:

in.NN; neural.param.NN; alphabet.NN

and three input files for the bp-algorithm:

bp.str, bp.tem, *.pat

discussed in the following section.
The in.NN file contains information needed by the simple genetic algorithm. The neural.param.NN file contains information needed by the semantic changing part of the algorithm.
The alphabet contains the alphabet of the higher level (symbolic) coding. You should stick to the given letters of the alphabet unless You want to write a parser (currently in the SRC/neural_bp_link.c file) that translates a different higher genetic coding into network files used by the bp. Nevertheless You may use any combination of the letters and place them in the alphabet as an additional letter.

Example of in.NN (You can find a detailed explanation in the genesis.txt file):

Experiments	=	1	#	number of independent experiments
Total Trials	=	400	#	total number of individuals evaluated during one experiment
Population Size	=	20	#	size of population; NOTICE: up to two times that number of individuals are evaluated each generation due to table adaption
Structure Length	=	60	#	bitstring length of an individual (lower level coding; here: 60 bits)
Crossover Rate	=	0.8	#	probability of crossover (here: 80%)
Mutation Rate	=	0.05	#	probability of mutation (here: 5%)
Generation Gap	=	0.9	#	amount of children that are entering the next generation (here: 90% of the new generation are children and 10% randomly picked parents.) A good value is usually >= 0.9.
Scaling Window	=	0	#	keep it 0; see genesis.txt
Report Interval	=	20	#	print statistics to the out.NN file each 20 evaluations of individuals; keep this number equal to the Population Size and You will get statistical information for each generation.
Structures Saved	=	20	#	how many of the best individuals of a generation You would like to save (each generation)
Max Gens w/o Eval	=	1	#	keep it 1; number of generation without evaluating the individuals
Dump Interval	=	1	#	save a generation after each “Dump Interval” generations in dump.* files; 1 = each generation.
Dumps Saved	=	20	#	maximal number of dumps; if You want to save all generations make it larger than: “Total Trails”/“Population Size”.
Options	=	aCe	#	more options are available; read genesis.txt; The options here aCe are advisable to have as a minimum. a: evaluate all individuals each generation, even if they have not changed. C: write statistical information to out.NN. e: use the elitist strategy, the best individual is automatically forwarded to the next generation. D: display information on the monitor; good if You do not want to run the program in the background mode.
Random Seed	=	183	#	a random number (seed); change it for each start of the program or You might obtain highly similar results.
Rank Min	=	0.75	#	NOT USED; for rank based selection, read genesis.txt.

Example of neural.param.NN:

Alphabet_File:	alphabet.NN	name of file containing the alphabet
Bit_Per_Symbol:	6	how many bits of the bitstring individual are translated into one translation table entry. IMPORTANT: This parameter sets the size of the translation table, the size is always $2^{Bit\_Per\_Symbol}$, the table contains entries with addresses from 00...0 to 11...1.
Neural_Seed:	453498756	initial number for the random number generator, change for each independent run.
Update_Table_ Spaces_Percent:	0.5	amount of table spaces to be modified during each adaption in % (here: 0.5 = 50% = 32). IMPORTANT: You have to leave a minimal amount of table spaces unchanged in order not to loose the best individuals of the current generation. To calculate how many places are maximally accessed by one individual You have to do the following calculation: "Structure Length" (defined in in.NN)/ "Bit_Per_Symbol" = minimal amount of table spaces that should be left unchanged. However it is more secure to leave at least twice the minimal amount unchanged.
Update_Table_ Mutation:	0.1	mutation probability for selected modules that enter the translation table (0.1 is a high mutation rate, You might want to set it lower; remember 0.5 is random !)
Gaussian_Max:	0.1	VERY IMPORTANT PARAMETER !!!! sets the preferred length (in % of the length of the donor symbol string) of substrings that will be selected from the donor individuals to enter the translation table during table adaption. Usually You want to set the parameter in a way so that the individuals (higher level/ symbolic coding) are neither forced to grow nor shrink. To achieve this You first have to calculate the average length of an alphabet letter (in the case of the simple alphabet: U,>,<,c, - this is 1). This average length You have to divide by ("Structure Length" (defined in the in.NN file)/"Bit_Per_ Symbol") to obtain the right value for the "Gaussian_Max" parameter. (here: Gaussian_Max = 1/(60/6) = 0.1) Setting the parameter to a higher value forces the individuals to grow and setting it to a lower value forces them to shrink.

Gaussian_Variance:	0.01	the enforcement of the preferred length; the preferred length is defined by a Gaussian and this parameter here defines its variance. The smaller the variance the less probable the individuals will change in size, the larger the variance the more likely they will change in size and the less the preferred length is enforced.
Input_Units:	11	number of input units of the neural network
Output_Units:	11	number of output units of the neural network
SaveInitialAlphabet:	0	0 do not save the initial alphabet, 1 save the initial alphabet
Init_Alphabet_File:	neural.alph_distribution	name of the file the initial alphabet will be saved to
Init_Table_File:	NEURAL_TABLES/ neural.table.initial.NN	file of initial translation table
Last_Table_File:	NEURAL_TABLES/ neural.table.last.NN	file the last translation table is saved to
Trace_Table_Step:	1	save a translation table after each Trace_Table_Step generations
Save_Table_Counter:	0	save translation tables starting with generation Save_Table_Counter
Table_File:	NEURAL_TABLES/ neural.table.NN	filestem for saved translation tables; full names are: neural.table.NN.<experiment>.<generation>
Last_Generation_File:	LAST_GENERATIONS /neural.last_generation.NN	file the last generation is saved to
BP_Startup_File:	BP_FILES/bp.script	shell script that starts the bp
BP_Network_File:	BP_FILES/bp.net	network input file for the bp
BP_Result_File:	BP_FILES/bp.log	result file created by the bp and set in BP_FILES/bp.str
Desired_Quality:	0.0	stop evolution if the Desired_Quality is reached (best quality is 0.0)

Example of alphabet.NN:

Letters of alphabet	Percentage	#	predefined line
———————	—————	#	predefined line
U	45%	#	letters of the alphabet
>	25%		followed by a percentage
<	equal		that states how many percent
-	equal		of the table spaces the
c	equal		letter should occupy after the
			initialization of the table.
			equal - means the letters are
			distributed equally over the
			remaining table entries.
			NOTICE: Do not try to fit more
			than 100%. The first letters
			with an "equal" occurrence may
			occur once more than the
			last ones.

The genetic engine yields three major outputs:

1. out.NN
2. dump.*
3. NEURAL_TABLES/neural.table.NN.<experiment>.<generation>

The out.NN file contains statistical information about the evolutionary run. You can find the explanation describing the file format in the genesis.txt file. The most important data are the first column, telling You the generation number, the first but last column telling You the quality of the best individual found till that generation (NOTICE: the best individual might be lost during one generation due to table adaption) and the last column, which contains the average performance of the entire population at the given generation. IMPORTANT: The best quality is always 0.0, qualities may range from 0.0 to +MAXDOUBLE.

The dump.<generation> files contain the bitstring population for the generations specified by the extension (see also genesis.txt) and additional information about that generation. NOTICE: Do not run more than one experiment at a time and do not set Experiments in the in.NN file to more than 1 as the dump.* files will be overwritten by a second experiment.

The NEURAL_TABLES/neural.table.NN.* files contain the translation tables for the generations indicated by the last file extension. After the header the file contains the table, where each table space is located on one line starting with the table space for 00...0 and ending with 11...1 (binary coding). NULL means that the table space is empty (NULL–string).

To translate the output files into the higher genetic coding and/or the network format for the bp see the section on TOOLS.

3. THE BACKPROPAGATION ENGINE
The bp is described in the book "Explorations in distributed parallel processing", D.E. Rumelhart, MIT Press, 1988 and will therefore not be described here.

The only additional file needed is the bp.script that starts the bp with the input files bp.tem and bp.str.

Edit bp.str to set all input parameters for the bp and/or additionally bp.tem to edit the output of the bp.

4. THE TOOLS
There are two tools:

best_individuals and individuals_to_bp_matrix

best_individuals - converts the low level (binary) genetic coding into the high level coding. type: best_individuals "amount_of_best_individuals(e.g. population size)" in.NN NEURAL_TABLES/neural.table.NN.1.x dump.x > "higher_level_coding_file" The "higher_level_coding_file" will contain the "amount_of_best_individuals" best individuals with their quality.

individuals_to_bp_matrix - converts the file created by best_individuals into network files that can be used with the bp. NOTICE: You will have to rename the files to *.net files for the bp to accept them!
type:
individuals_to_bp_matrix "higher_level_coding_file" neural.param.NN "output_filename_stem"
Files called "output_filename_stem".* will be created, each containing a bp network. The first network coded genetically in the "higher_level_coding_file" is saved in "output_filename_stem".0, the second in "output_filename_stem".1 etc.

5. SOURCES
The interesting sources are:

SRC/neural_fitness.c - for changing the fitness function

Bibliography

[1] Boers, E.J.W. and Kuiper, H.
Biological metaphors and the design of modular artificial neural networks
Master's Thesis, Leiden University, The Netherlands. [In ftp.cs.tu-berlin.de: pub/doc/neuroprose/boers.biological-metaphors.ps.gz]

[2] Brause, R.
Neuronale Netze
Teubner, Stuttgart 1991.

[3] De Jong, K.A.
An analysis of the behavior of a class of genetic adaptive systems
Doctoral dissertation, University of Michigan, Dissertation Abstracts International 36(10), 5140B. (University Microfilms No. 76-9381)

[4] Harp, S.A., Samad T. and Guha A.
Towards the genetic synthesis of neural networks.
In J.D. Schaffer, Editor, Proc. of the Third Int'l Conf. on Genetic Algorithms and Their Applications, pp. 360–369.
Morgan Kaufmann, San Mateo, CA, 1989

[5] Harp, S.A., Samad T. and Guha A.
Designing application–specific neural networks using the genetic algorithm.
In D.S. Touretzky, Editor, Advances in Neural Information Processing Systems 2, pp. 447–454, Morgan Kaufmann, San Mateo, CA, 1990

[6] Harp, S.A. and Samad T.
Genetic synthesis of neural network architecture.
In L. Davis, Editor, Handbook of Genetic Algorithms, pp. 203–221.
Van Nostrand Reinhold, New York, NY 10003, 1991.

[7] Hecht-Nielsen, R.
Neurocomputing
Addison-Wesley 1990.

[8] Hertz, J., Krogh, A. and Palmer, R.G.
Introduction to the theory of neural computation
Addison–Wesley Publishing Company, 1991.

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

[10] Goldberg, D.E.
Genetic Algorithms in search, optimization and machine learning.
Addison–Wesley, Reading, 1989.

[11] Gruau, F.
Neural Networks Synthesis using Cellular Encoding and the Genetic Algorithm
[PhD. thesis in 140.77.1.11: pub/Rapports/PhD/PhD94–01–E.ps.Z], 1994

[12] Kitano, H.
Designing neural networks using genetic algorithms with graph generation system.
Complex Systems, 4:461–476, 1990.

[13] Kramer, A.H. and Sangiovanni–Vincentelli, A.
Efficient Parallel Learning Algorithms for Neural Networks.
In Advances in Neural Information Processing Systems I (Denver 1988), ed. D.S. Touretzky, pp. 40–48. San Mateo: Morgan Kaufmann. 1989.

[14] Koza, J.R.
Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems
Stanford University Computer Science Department Technical Report STAN–CS–90–1314. June 1990.

[15] Koza, J.R. and Rice, J.P.
Genetic generation of both the weights and architecture for a neural network.
In International Joint Conference on Neural Networks 1991 (IJCNN 1991), pp. II-397 – II-404

[16] Lin, E.C.C.
Bacteria, plasmids and phages: An introduction to molecular biology
Cambridge, Mass.: Harvard University Press, 1984.

[17] Marti, L.
Genetically Generated Neural Networks I: Representational Effects
CAS/CNS–TR–92–014, 1992;
Boston University Center for Adaptive Systems;
111 Cummington Street Boston, Ma 02215;
[In ftp.cs.tu-berlin.de: pub/doc/neuroprose/marti.ga1.ps.gz]

[18] Miller, G.F., Todd, P.M. and Hegde, S.U.
Designing neural networks using genetic algorithms.

In J.D. Schaffer, Editor, Proc. of the Third Int'l Conf. on Genetic Algorithms and Their Applications, pp. 379–384.
Morgan Kaufmann, San Mateo, CA, 1989

[19] Mjolsness, E., Sharp, D.H. and Alpert, B.K.
Scaling, machine learning and genetic neural nets.
Advances in Applied Mathematics, 10:137–163, 1989.

[20] Merrill, J.W.L. and Port, R.F.
Fractally configured neural networks.
Neural Networks, 4:53–60, 1991.

[21] Montana, D. and Davis, L.
Training feedforward neural networks using genetic algorithms.
In Proc. of Eleventh Int'l Joint Conf. on Artificial Intelligence, pp. 762–767. Morgan Kaufmann, San Mateo, CA, 1989.

[22] Ossen, A.
Zur Modularisierung und Interpretierbarkeit Neuronaler Netze
Dissertation, Berlin 1990.

[23] Prechelt, L.
PROBEN1 — A Set of Benchmarks and Benchmarking Rules for Neural Network Training Algorithms.
[In ftp.ira.uka.de: /pub/papers/techreports/1994/1994-21.ps.Z]

[24] Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T.
Numerical Recipes.
Cambridge: Cambridge University Press, 1986.

[25] Rechenberg, I.
Evolutionstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution
Frommberg–Holzboog, Stuttgart, 1973.

[26] Rojas, R.
Theorie der neuronalen Netze - Eine systematische Einführung
Springer-Verlag, Berlin Heidelberg 1993.

[27] Rueckl, J.G., Cave, K.R. and Kosslyn, S.M.
Why are 'what' and 'where' processed by separate cortical visual systems ? A computational investigation.
In Journal of cognitive neuroscience,1 ,pp. 171 – 186, 1989.

[28] Rumelhart D.E., Hinton G.E., Williams R.J.
Learning internal representations by error propagation
In: D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations, MIT Press 1986.

[29] McClelland, J.L. and Rumelhart D.E.
Explorations in parallel distributed processing
MIT Press, 1988.

[30] Salomon, R.
Verbesserung konnektionistischer Lernverfahren, die nach der Gradientenmethode arbeiten
Dissertation, TU Berlin 1991.

[31] Sałustowicz, R.P.
Genetische Algorithmen zur Konstruktion Neuronaler Netze – eine vergleichende Studie
1994, unpublished pre–diploma thesis, Technical University of Berlin.

[32] Schwartz, J. T.
The new connectionism - Developing relationships between neuroscience and artificial intelligence
In: S.R. Graubard (ed.), The Artificial Intelligence Debate, MIT Press 1988, pp. 123–141.

[33] Whitley, D., Starkweather, T. und Bogart, C.
Genetic algorithms and neural networks: optimizing connections and connectivity.
Parallel Computing. 14 (August 1990) 3. pp. 347–361.

[34] Yao, X.
A Review of Evolutionary Artificial Neural Networks
[In ftp.cs.tu-berlin.de: pub/doc/neuroprose/yao.eann.ps.gz]