Synthesis and Performance Analysis of Multilayer Neural Network Architectures

W. Schiffmann, M. Joost, R. Werner
University of Koblenz
Institute für Physics
Rheinau 3–4
D-5400 Koblenz
e-mail: schiff@infko.uni-koblenz.de

Technical Report 16/1992

Contents

1 Introduction

It is obvious that *natural* Neural Networks aren't fully interconnected. There are at least 10 to $100 \cdot 10^{10}$ neurons and only approximately $100 \cdot 10^{12}$ synapses. If the brain would be fully interconnected the number of synapses must be in the range of 10^{20}. Neurophysiological investigations confirm that the various brain functions emerge from hierarchical structured neural paths.

For example the stages of the visual path were studied by [Hubel, 1988]. They presented visual stimulies to cats and analysed the neural response to those patterns in different brain areas. Different functional blocks were identified which extract hierarchically ordered features of the input patterns. The so–called *On–Off*–Cells of the retina perform an edge enhancement and compensate illumination effects. *Simple* Cells in the striate cortex recognize specific dark–light patterns under various orientations. Similarily *complex* Cells response to the same patterns but only if those patterns are moved in a specific direction within the receptive field of the cell.

Because it is only possible to register superimposed responses of multiple cells, the results of Hubel (and his colleague Wiesel) delineate a "fuzzy" picture of early visual processing. Nevertheless, they illustrate the importance of the neural architecture for solving a specific task.

In contrast to such neurophysiological facts most *artificial* neural networks are not structured. Their architecture is determined by the intuition of a human designer who has to decide how many formal neurons — called *units* in the following — should be used and how these units should be interconnected. The network architecture can be described as directed graph. The units are the nodes and the connections are the edges. Finding problem–adapted network architectures is equivalent to optimizing the topology of the underlying network graph.

In this paper we present various approaches for automatic topology–optimization of backpropagation networks. First of all, we review the basics of genetic algorithms which are our essential tool for a topology search. Then we give a survey of backprop and the topological properties of feedforward networks. We report on pioneer work in the filed of topology–optimization. Our first approach was based on evolutions strategies which used only mutation to change the parent's topologies [Schiffmann, 1989], [Schiffmann and Mecklenburg, 1990] and [Schiffmann et al., 1990]. Now, we found a way to extend this approach by an crossover operator which is essential to all genetic search methods.

In the following this genuine genetic approach is presented together with interesting simulation results. In contrast to competing approaches it allows that two parent networks with different number of units can mate and produce a (valid) child network, which inherits genes from both of the parents. We applied our genetic algorithm to a medical classification problem which is extremly difficult to solve. The performance with respect to the training set and a test set of pattern samples was compared to fixed network topologies. Our results confirm that the topology optimization makes sense, because the generated networks outperform the fixed topologies and reach classification performances near optimum.

Working with optimized architectures offers benefits both for software simulations and for hardware implemtation of artificial neural networks. On one hand, training is accelerated because fewer connections must be trained per epoch. The same holds for the real time behaviour during recall. Therefore we can postulate that optimizing the architecture is as well important as finding new learning rules. In particular, optimized architectures could be used for VLSI–implementation, because the reduced number of connections simplifies the layout of the chips. We think that much more efforts in topology–optimization must be done in order to make such *Neuro–Chips* feasible.

2 Genetic Algorithms

2.1 Overview

Genetic algorithms (GAs) are used to solve optimization problems. We must be aware that several types of GAs have evolved over the years. The main common feature of GAs is that they sample a *performance* or *fitness* function which depends on variable and unkown parameters. Two types can be distinguished:

1. continuous and
2. discrete ones

2.2 Continuous Parameters

In the first case the GA is mainly used to optimize various kinds of functions even nonlinear or non steady functions. This flexibility makes GAs applicable to a wide range of optimization problems which cannot be solved by other numerical methods. For example, gradient descent is not possible if the performance function cannot be derived with respect to the parameters.

Because neural networks usually contain a lot of nonlinear processing units, the resulting performance is nonlinear as well. The parameters of the neural network are the connection strenghts between the units. These weights must be adjusted with the help of training data. The network should approximate and extrapolate (e.g. generalize from) the training data as good as possible. The most popular learning algorithm uses a quasi–nonlinear activation function und implements a (fast) gradient descent (see section 3). Nevertheless, there are some approaches which adjust the connection weights by means of GAs. We return to such net training GAs later and report about publications concerned with that approach.

2.3 Discrete Parameters

Here, the parameters can olny adopt on several discrete values. The solution of the problem is a certain *combination* of discrete parameter values. Prominent and often used combinatorial optimization problems are the Traveling Salesman Problem (TSP) and the Graph Partitioning Problem (GPP). A main feature of those problems is that they are NP–complete. There is no efficient algorithm to solve them if the problem size is increased. Instead, the computational effort grows exponentionelly with the problem size.

By sampling the performance function GAs provide approximate solutions with reasonable effort. This search is both random as well as directed. In contrast to deterministic but intractable algorithms GAs cannot garuantee to find the absolutely best solution to a combinatorial problem. But if the population size and the genetic operators are appropriately choosen they find near optimum solutions.

Like TSP and GPP, topology optimization is a combinatorial problem. We are looking for a well suited graph of units and connections which is adopted to both the training and test data as well as to the learning algorithm used. We will discuss this kind of GA in more detail in section 4.

2.4 Sampling the fitness function

As already mentioned GAs adjust parameters by sampling a fitness function. New and maybe better parameter values can emerge by the application of genetic operators:

1. Mutation
2. Crossover
3. Recombination

2.4.1 Mutation

Mutation is the most simple genetic operator. It randomly changes the value of a parameter with probability p_m. This depends on the representation (coding) scheme of the parameters. Often, the parameters are binary coded. A bit flip in a randomly determined position of the *genotype string* is applied. This string contains all the parameters of the actual problem. After interpretation of that string the corresponding parameter values (often called chromosomes) define a *phenotype* which can be scored according to a given fitness function.

If the (continous) parameters are represented immediately by floating point numbers, mutation is equivalent to scattering one or more parameters around its old values. This can be achieved by a multivariate gaussian probability distribution with a given standard derivative σ (for all dimensions).

2.4.2 Crossover

Crossover is used to mix the genetic material of two parents. Usually the genotype strings have the same length (number of chromosomes). Crossover takes place with probability p_c and is divided into three substeps:

1. Choose two mating parents
2. Choose a common crossover point and
3. Exchange the chromosomes of the two genotype strings

If a binary representation scheme is used the crossover point can be placed within chromosomes (e.g. a floating point number). In common, there can be more than one crossover point.

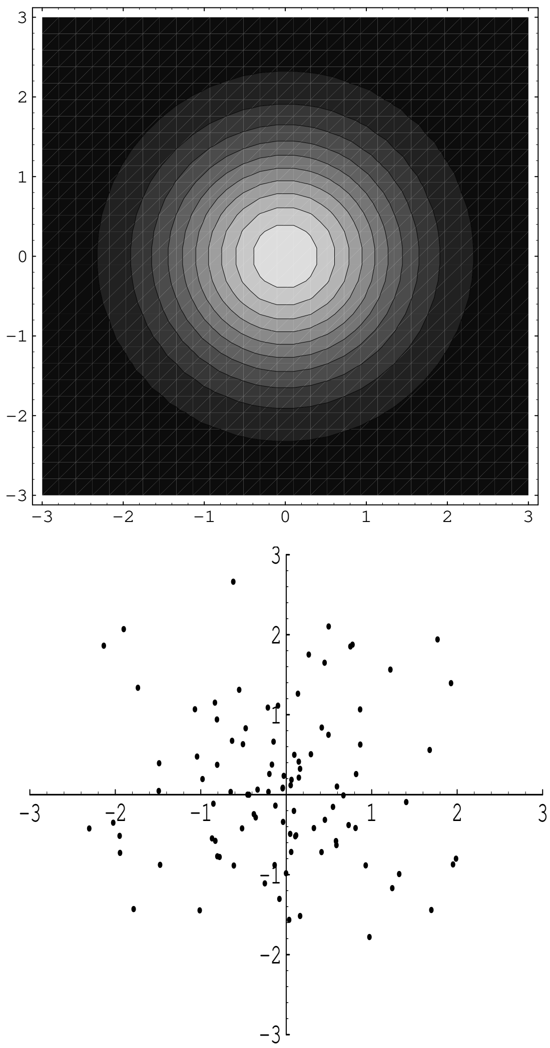

Figure 1: Mutation in a two dimensional space around a specific parameter vector. **Above:** Combined contour and density plot of the gaussian probability distribution with standard derivative $\sigma = 1.0$ (both directions). The brightness of the a specific point corresponds to the probability of an offspring. **Below:** Concrete distribution of 100 points

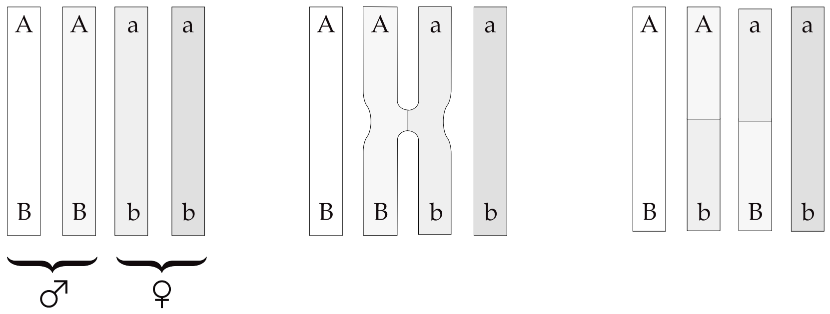

Figure 2: Crossover operator

2.4.3 Recombination

This genetic operator can be thought of as a special case of multiple crossover with the crossover points adjusted to the borders of chromosomes (binary coding). It is applied with probability p_r.

2.5 Mutational Sampling

If we choose $p_c = p_r = 0$ new (better) parameters can emerge only by chance. This special case of a GA is often called *evolution strategy* although each GA is evolutionary. In the following we refer to it by the appreviation MA for *mutational* algorithm. MAs are very slow if there is no adaptation of the step width, e.g. the standard derivation is a gaussian probability distribution is used. The convergence can be accelerated by appropriate heuristics in order to adapt the scatter area to the local properties of the search space. For example, different standard derivatives σ_i for the parameters x_i are used and updated by the statistical properties of the preceding generation (see [Rechenberg, 1973] and [Schwefel, 1976]).

2.6 Genetic Sampling

As shown in Figure 3 new sampling centers are provided by the combination of two parent strings. If these strings are close together the scatter area of the new sampling centers is small. Otherwise, if the parent strings are far away in the parameter space the scatter area is large. In this way GAs adapt their step width by combining the genetic material of the population. No heuristics are needed to control step width adaptation. Common substrings of genotypes are maintained. They are called *building blocks*. Different substrings are changed by searching new and closer ones which simultaneously improve the fitness function.

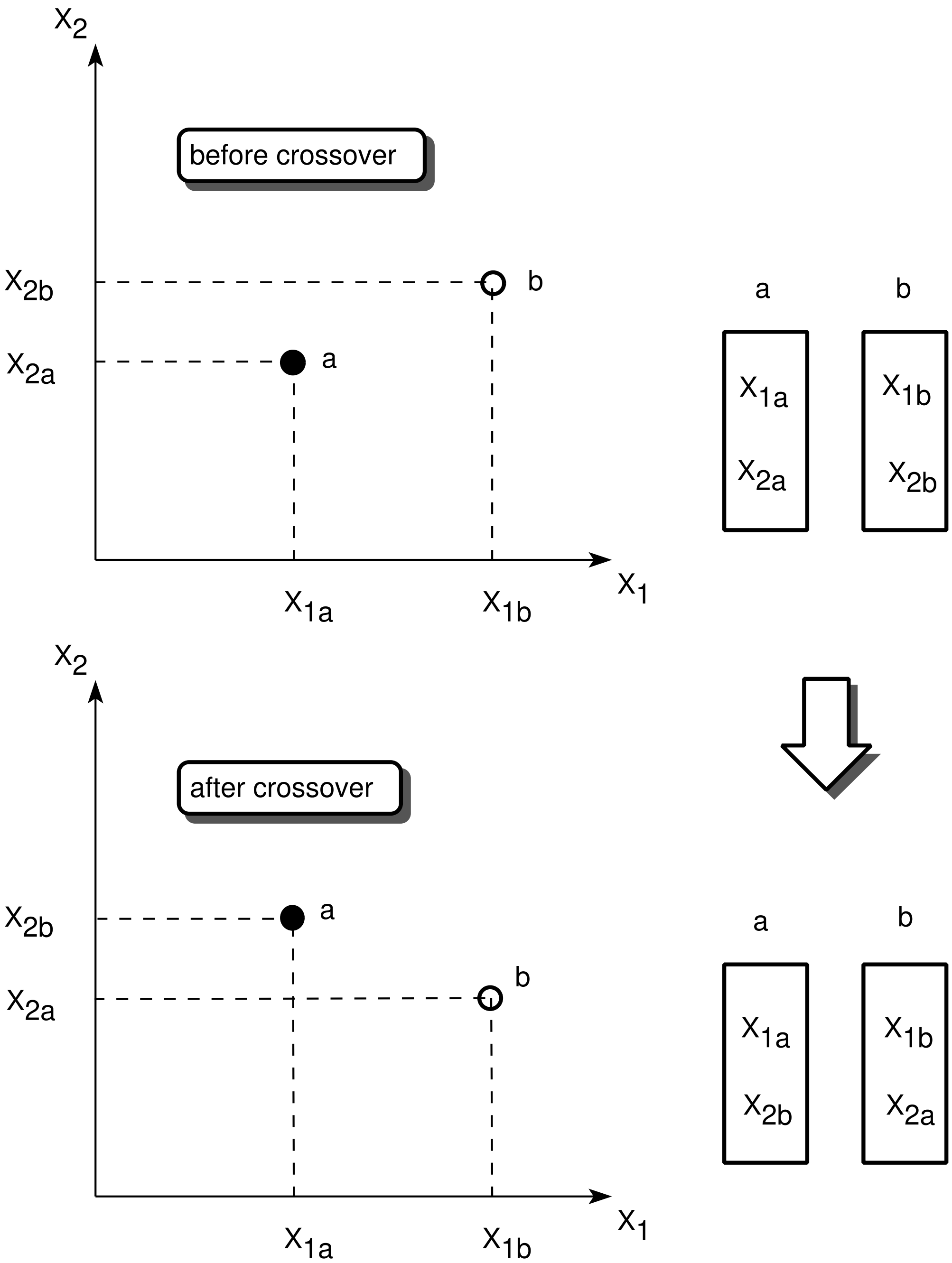

Figure 3: Crossover operator in twodimensional parameter space

2.7 GA Outline

We want to delineat the outline of a basic GA. First of all we have to encode our parameters in an appropriate representation. Then the following basic steps are executed:

1. Initialize one (more) population(s) of parent strings
2. Select two parents
3. Mutate selected chromosomes
4. Combine the mutated parent strings (crossover, recombination)
5. Compute the fitness of the offsprings
6. Continue with step 2 until a given performance is achieved

MAs are a special case of the given scheme: just one parent is choosen in step 2 and step 4 is omitted. An additional local hill climbing can take place between step 4 and 5. Because GAs are well suited to find just a crude solution area, local hill climbing (e.g. a gradient method) does a fine tuning of the parameters.

2.8 Control parameters

The given GA outline depends on a lot of control parameters. If we don't find appropriate values for it, the performance of the GA degrades. For example, if the mutation rate p_m is too large we jump aimlessly through the parameter space. If it is too small, sampling is controlled only by the differences of the parent strings. If there is little variablility within the population, progress get stuck. The same considerations hold for the standard derivation σ in the case of a floating point representation. Other control parameters are the size of the population, the number of crossover points and s.o.

2.9 Selection scheme

Two different selection schemes can be distinguished: *roulette wheel* selection and *breeding*. In the first case the selection probability of a parent string is proportional to the normalized fitness of that string related to the members of the population. When breeding or *truncated selection* is choosen only the *best* members of a population were used as parents for the next generation. We used this scheme in our early work on the actual subject [Schiffmann et al., 1990]. It should be noted that another two control parameters are required which specify *how many* best parents should survive and how many offsprings will be produced. For the present paper we used the roulette wheel selection scheme.

2.10 Population size and organization

Another important feature of GAs is concerned with the population size and the organization of the population(s). We can choose one centrally organized population with many members or we can distribute the genotypes over many small populations. The last method is especiallly recommendable if we use parallel computers. It also requires a distributed (local) selection scheme but a global fitness function. These *parallel* GAs (PGAs) were introduced by [Mühlenbein, 1992] and applied to the TSP and GPP.

3 Multilayer Perceptrons

As [Minsky and Papert, 1969] have shown the XOR–problem cannot be solved without a hidden layer. [Hornik et al., 1989] proved that every function can be approximated by a neural network with just *one* hidden layer. By adding "enough" hidden units the approximation error can be made as small as required. This can be compared with a look–up table which stores one output value/vector per hidden unit. But, this approach doesn't take into account that the network should be able to generalize. It just tries to fit the training data. If only connections between adjacent layers are present the effect of multiple hidden layers can be visualized in pattern space. Examples of such networks with 1,2 and 3 hidden layers are given in [Longstaff and Cross, 1987]. One can imagine the operation of a multilayer network only for those special cases.

3.1 Training multilayer perceptrons

It is out of the scope of this paper to give an detailed representation of the backprop algorithm. It can be found elsewhere [Rumelhart et al., 1986]. Here, we summarize the main ideas and essential formulas of backpropagation (BP). First of all we need a training pattern set which consists of pairs of input and output vectors. This kind of learning is called *supervised* learning in contrast to *unsupervised* learning where only input vectors are presented. Second a topology must be given on which BP operates. For each training pattern the connection weights are modified —by a small amount— in direction of the negative derivative of the squared error with respect to the corresponding weight. In this way a gradient descent on the (mapping) error surface is implemented. The width of each update step is controlled by the so–called learning rate η.

$$
\begin{aligned}
E_p &= \frac{1}{2}\sum_k E_k = \frac{1}{2}\sum_k (t_k - o_k)^2 && (1)\\
w_{ij}(t+1) &= w_{ij} - \eta \frac{\partial E_p}{\partial w_{ij}(t)} && (2)
\end{aligned}
$$

where o_k are the network outputs when the pattern p is applied and t_k are the desired outputs. BP operates in two phases. In the *forward–pass* the activation pattern of the input vector is propagated through the network and produces an output vector. In the *backward–pass* learning takes place. Differences between the desired and the actual outputs are propagted back ($\leadsto$ name) from output to input layer. The error of given unit is used to adjust its weights according to the *generalized delta–rule*:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \cdot \delta \cdot o_i \tag{3}$$

Suppose we use units with a sigmoid activation function

$$o(net) = \frac{1}{1 + e^{-net_j}} \tag{4}$$

with $net_j = \sum_i w_{ij} \cdot o_i$
Then we update the weights of the output units o_k by choosing

$$\delta_k = o_k(1 - o_k)(t_k - o_k) \tag{5}$$

For a hidden unit o_x we must sum up the weighted errors of all successive units. The resulting sum is again scaled with the derivative of the sigmoide activation function at the operation point.

$$\delta_x = o_x(1 - o_x) \sum_{i \epsilon \mathbf{S}(x)} \delta_i \cdot w_{xi} \tag{6}$$

where $\mathbf{S}$(i) denotes the set of all successors of the unit i. Note that the maximum value of $o(1 - o)$ is 0.25 and that this value is raised to powers of the number of the hidden layers. Thus, the more units an error passes through from output to input layer the weaker is its influence on the weights in the first layers.

3.2 Local minima

Like every gradient descent method BP is faced with the problem of local minima. If the learning rate isn't too big BP always goes to the next local minimum (with respect to its initialized connection weights). Even though there is little chance to reach the global minimum almost all local minima have reasonable low errors [Hecht–Nielsen, 1989].

3.3 Speed of convergence

In order to get a quick and straightforward convergence of the gradient descent a lot of learning rate adaptation schemes have been proposed. The simplest method to adapt the learning rate η was introduced by [Rumelhart et al., 1986]. The momentum term $\alpha \Delta w_{ij}(t-1)$ has two effects:

1. oscillations in the proximity of a local minimum are damped and
2. the convergence is accelerated in wide plateaus where the gradient is very small

Other more sophisticated approaches to speed up BP can be found in [Chan and Fallside, 1987], [Jacobs, 1988] or [Silva and Almeida, 1990]. A detailed comparison of those approaches is now available from [Schiffmann et al., 1992].

3.4 Need for automatic topology search

BP is working on a given network architecture which is made up by a specific partition of units over layers and by a connectivity pattern. Choosing an appropriate topology for a given problem depends on personal experience of the human designer. Almost always a fully interconnected architecture is used. But what is the optimal number of units und their organization into layers?

With exception of some simple task, e.g. the XOR–problem, humans cannot foresee the optimal network topology. Thumb rules like *the harder the problem the more units you need* are of little practical use to the design problem. Manual network design is something of a black art [Miller, et al., 1989] and it can be assumed that most of such network designs are not optimal. In [Schiffmann and Mecklenburg, 1990] the impact of network topology on the speed and performance of BP trained networks was demonstrated. In order to adapt the network topology to the problem at hand we proposed an *automatic* design procedure which is based on *Genetic Algorithms* (GAs) (see also [Schiffmann, 1989]).

Later on, we evaluated the performance and generalization behaviour of automatic generated network architectures [Schiffmann et al., 1990] with respect to character recognition tasks. While our first approaches used just mutation to modify the network topology we are now able to cross two architectures.

3.5 Possible partitions

Suppose a wizard would tell us that h hidden units and l hidden layers are necessary for a given problem. Now, we have to distribute the hidden units into these layers. We will derive a recursive formula which allows to compute the number of possible partitions $p(h, l)$. It is evident that

$$p(h, l) = 1 \quad \text{if} \quad l = 1 \quad \text{or if} \quad h = l \tag{7}$$

If $h < l$ then $p(h, l) = 0$ because empty partitions are not reasonable. If $h > l$ we get

$$p(h, l) = p(h - 1, l - 1) + p(h - 1, l) \tag{8}$$

By this formula we can compute the number of possible partitions from that number of a less complex architecture with $h - 1$ units. The first term counts the number of partitions if the boundary of an additional layer seperates the additional unit itself. The second term means that the additional unit is placed into the last hidden layer of a less complex architecture which has already l layers. As table 1 illustrates the number of possible partitions —even for moderate numbers of hidden units— increases to astronomical values.

3.6 Possible connections

If we have choosen a specific partition we are faced with the problem of optimizing connectivity. For h hidden units the number of possible connections is limited by two extreme topologies which are fully interconnected from input layer (m units) to output layer (n units):

1. A topology that has as much as possible hidden layers with one hidden unit each. We will refer to that topology as TALL.
2. A topology that has just one hidden layer. It forms a look–up table and we will refer to it as WIDE.

l/h	10	20	30	40	50	60	70	80	90	100
1	1	1	1	1	1	1	1	1	1	1
2	9	19	29	39	49	59	69	79	89	99
3	36	171	406	741	1176	1711	2346	3081	3916	4851
4	84	969	3654	9139	18424	32509	52394	79079	113564	156849
5	126	3876	23751	82251	211876	455126	864501	1502501	2441626	3764376

Table 1: Number of possible partitions if h hidden units should be distributed over l (non empty) layers

While the TALL–architecture contains

$$C_T = \frac{h^2 - h}{2} + h \cdot (m + n) + m \cdot n \tag{9}$$

connections, the WIDE–architecture has only

$$C_W = h \cdot (m + n) + m \cdot n \tag{10}$$

connections. Table 2 compares these two topologies to each other for various values of hidden units. Additionally the fraction of C_W/C_T is given. It can be used as a measure of connectivity.

In most practical applications neither the TALL– nor the WIDE–architecture will be best suited. Thus, we have to find a connectivity pattern which adapted to a specific task. Because we (humans) cannot comprehend the effects of general modifications in topology, we have to provide methods for topology optimization which operate automatically.

h	C_T	C_W	C_W/C_T
1	87	87	1.000000
25	963	663	0.688474
50	2488	1263	0.507637
75	4638	1863	0.401682
100	7413	2463	0.332254
200	24763	4863	0.196382
300	52113	7263	0.139370

Table 2: Number of possible connections and minimum connectivity for the two extreme architectures (m=21, n=3, see also section *Simulation Results*)

4 Applications of GAs to Neural Networks

It should be noted that GAs can be applied to neural networks in two different ways:

1. Optimizing connection weights
2. Optimizing network topology

4.1 GAs for weight adjustment

In the first case the GA works at continuous parameters. Results concerned with this approach can be found in [Whitley, 1988 and 1989], [Heistermann, 1989 and 1990], [Montana and Davis, 1989] and [de Garis, 1990]. An exciting discussion of evolutionary training methods is provided by [Weiss, 1990]. It can be summarized that the mentioned approaches differ mainly in three ways:

1. number representation
2. genetic operators used and
3. parent–offspring replacement strategies

If the number of connections is high it takes a lot of time to train a network by means of GAs. As simulations show, just less complex networks (<50 units) can be trained by this method. The speed of convergence can be increased if the rate of mutation is inverse to the diversity in the population of networks [Whitley and Hanson, 1989].

GAs for weight adjustment implement a parallel search in weight space. Thus, they are able to find approximative solutions in short time. In contrast, GAs have trouble to get an exact solution. In order to solve this problem, one can combine GAs with BP. The starting weight vector for BP is determined by a preceding GA training run. See [Kitano, 1990] for details of that approach.

4.2 GAs for topology optimization

Here, learning is done by BP or any other well known learning procedure. Because discrete parameters must be optimized the performance function is undifferentiable. Thus, gradient methods are not applicable. As we have seen the search space of all possible network architectures is vast and noisy. Because it depends on the random initial weights the performance of a specific architecture must be viewed as a random variable. It is also *deceptive* and *multimodal*. These features are concerned with the effects of small changes of the parameters and the objective function. *Deceptive* means that similar network architectures can have different performance. On the other hand, different network architectures show similar performance. Hence, the search space is *multimodal*.

As pointed out in [Miller et al., 1989] such complex spaces cannot be explored efficiently be enumerative, random or even heuristic knowledge–guided search methods. In contrast, the adaptive features of GAs (building blocks, step–width control by crossover) provide a more robust and faster search procedure. Additionally, it is easy to speed–up the genetic search by means of parallel processing.

Before we discuss the differences between several approaches we want to give a basic GA for topology optimization which is common to all these approaches. It is assumed that two representations of the networks are distinguished:

1. *genotypes* which are modified by the GA's operators (mutation, crossover)
2. *phenotypes* which are trained by a conventional learning procedure (e.g. BP) used for performance evaluation or selection

The basic GA comprises four steps:

1. Initialize a population of random starting architectures
2. Select, cross(over) and mutate these architectures
3. Train the networks for a given number of epochs
4. Stop if desired performance is achieved; else proceed with step 2

4.3 Representation of genotypes

Roughly two basic representation schemes can be distinguished:

1. low–level genotypes
2. high–level genotypes

While the first one is transparent and easy to use, there are two variants of the second representation scheme. Low–level genotypes directly code the network topology. Each unit and each connection is specified separately. This "blueprint" approach is used by [Miller et al., 1989], [Schiffmann, 1989], [Dodd, 1990], [Schiffmann and Mecklenburg, 1990] and [Schiffmann et al., 1991].

High–level genotypes are more complex coded representations of network architectures. They can be further divided into *parametric* and *recipe* genotypes. Examples for the use of parametric genotypes are found in [Lehar and Weaver, 1987], [Harp et al., 1989] and [Merrill and Port, 1991]. Here, the networks are splited into modules of units which are specified by parameters and which are coupled by parametric connectivity patterns. Even though this representation is more compact and thus well suited to code large network architectures, it is difficult to choose the *relevant* parametric shapes. It should be noted that in using parametric genotypes the search space is confined to a specific subspace.

Similar considerations apply to the recipe genotype representation. Here, the architecture is specified by growth rules [Mjolsness et al., 1986–1987] or by sentences of a formal language [Mühlenbein and Kindermann, 1989]. The last approach is designates *genetic neural networks*. Even though these approaches are mostly biological plausible, one has to commit to primitives used for the rules. Further, the choice of specific primitives hardly influences the application of genetic operators.

Usually neural network applications are limited to network sizes of about 1000 units. Often, research applications are confined to smaller architectures (e.g. XOR– or encoder–problem). Thus, in order to evaluate the properties of GAs for topology optimization the low–level representation is well suited and sufficient. Because it is easy and straightforward to apply genetic operators to the blueprints, we decided to use it to code our network architectures.

4.4 Genetic Operators

As pointed out in [Weiss, 1990] the genetic operators must produce *correct* and *complete* offsprings. These requirements are easy to satisfy with blueprints. For example, it can be verified for our mutation operator which does one of the following topological modifications [Schiffmann et al., 1991]:

1. remove/add (feedforward) connections
2. add new hidden units with some random connectivity

Because each topology can be build up with these operations, the mutation operator is complete. Only hidden units and feedforward connections are added. Thus, all networks are correct multilayer perceptrons. Note that completeness —in the sense used here— doesn't imply that all possible architectures arise with equal probability. If the insertion points for new connections aren't constrained, networks with many hidden layers are more likely.

Our new crossover operator is described in a separate section. In contrast to the mutation operator the application of crossover can produce abrupt changes of architectures. It is an open question and there are contrary viewpoints if its unconstrained application is reasonable or not. One can argue that only similar architectures should be crossed in order to avoid abrupt changes. This viewpoint postulates that the surface of the fitness function is smooth and that small search steps are likely to succeed.

On the other hand, we can argue that successful abrupt topology changes indicate that functional sub–networks have been exchanged between two parent networks. From that viewpoint, the borders of these building blocks are recognized automatically by means of selection. Because it is difficult or even not possible to define similarity measures between two topologies, we promote the last viewpoint. We cross networks of *different* sizes and don't constrain the position of the crossover point within the blueprints of the parent networks.

4.5 Selection criteria

Selection can be guided by the following criteria or combinations of them:

1. Mapping error (training set)
2. Generalization behaviour (large test set)
3. Speed of training
4. Complexity of created networks (number of units/connections)

The criteria listed above can be combined in order to achieve arbitrary design objectives. For example, we have created architectures which have maximum accuracy with respect to the training data and which can be trained quickly. Note that the speed of training is measured indirectly. The networks of the population are trained for a given number of epochs and then a roulette wheel selection takes place. In contrast to our previous work [Schiffmann et al., 1990] we decided to use just one population for parent and offspring networks (see section 5). This implies that we also prefer small network architectures because these networks can be trained faster than big ones. Because it is difficult to define an objective function for an assessment of the generalization behaviour, we omitted it in our present work. It is subject of further research.

4.6 Blueprints and crossover

An efficient blueprint representation can be achieved by using a list where the connectivity of individual units is registered. In order to use this data structure together with BP, the list must be chained in both directions. For the sake of simplicity, this is not shown in Figure 4.

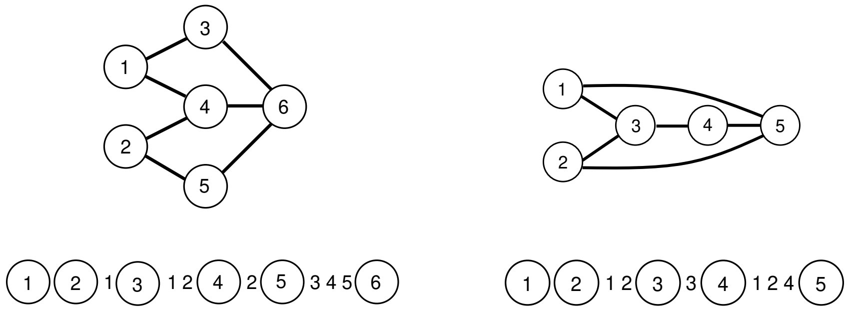

Figure 4: Two network architectures with its corresponding blueprint representations

Blueprint coding works in two steps. Starting with the input layer, the units are numbered. Then, for each of the units the numbers of its preceding units are registered in the list. Note that topologically equivalent architectures can have different blueprint representations.

Crossover is done by choosing a common cross point and joining two sub–lists of the blueprints. The cross point must be positioned before the last output unit of that network which has the lower number of units. If this constraint is satisfied it is guaranteed that the produced offspring operates on the same interface units (input and output) as its parents. So we get an *useful* network (see Figure 5). In this way, we have defined a *correct* crossover operator which is able to cross networks of *arbitrary* size.

In order to guarantee that the offspring isn't identical with one of the parent networks we must position the cross point behind the last input unit. Nevertheless we cannot guarantee that always *new* architectures arise. Even though we get useful networks it could happen that isolated or fixed hidden units arise. The output of *isolated* units isn't used by any other unit. *Fixed* units don't have connections to preceding units. Its weighted (constant) output can be substituted by the bias units and its connection weights. Both kinds of useless[1] units can be eliminated without affecting the networks functionality. In this way the shrunk offspring networks can become smaller than the smallest parent network. On the other side, it isn't possible that the number of units becomes larger than the biggest parent network. The number of connections can grow when the described crossover operator is applied.

[1]information is neither combined nor processed

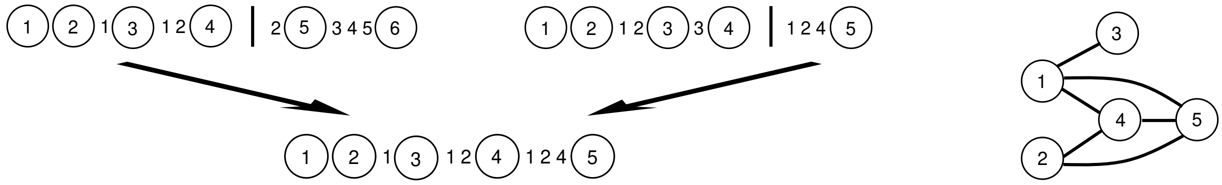

Figure 5: Application of our crossover operator to the networks in the previous Figure. The isolated unit 3 can be removed.

5 Simulation Results

5.1 Implementation

In order to accelerate the performance of the genetic algorithm it was implemented on a UNIX–LAN. All connected UNIX workstations can be used for the evolution of new architectures. The networks are produced and trained in parallel by independent programs running on these workstations. These processes – the so called evolution servers – communicate via sockets with a control program which runs on the users workstation. The control program offers a graphical user interface for starting and controlling the evolution servers. Here, all parameters for steering the evolution process and the training of the nets are adjusted. Most of these parameters are interactively variable during the evolution and are transmitted immediately to the evolution server.

The selection is made centrally. As soon as an evolution server has determined the quality of a net, it will be transmitted to the control program, which decides whether the developed net will be inserted into the population or not.

In our previous work parents and offsprings were separated in different populations. Out of m parent nets n offsprings (n$\geq$m) were made. Then the m best offsprings were selected to a new parent generation. This concept was given up. It is replaced by a selection which operates only on one population. A new net will be immediately included in the population and can directly take part in the further evolution process. So an existing topology advantage can spread out faster.

The most important advantage of this selection procedure is that smaller nets are preferred. As smaller nets are trained faster than larger ones, they have, in the long term, a better chance to come along.

A further difference to [Schiffmann et al., 1991] is that the trained weights of the parent nets are not transmitted any longer to the offsprings. Before training all weights of a offspring will be set by chance. So we obtain topologies which can be trained very fast during the given number of learning passes.

In detail, our topology optimization works as follows:

Control program:

- Selection of two nets by means of roulette wheel procedure. Good nets have a better chance to be selected than bad ones.

Evolution server:

- Crossing of the selected nets. The arising net is then mutated to a small degree.
- Initializing, training and testing of the net. The number of correct classification will be transmitted as quality measure.

Control program:

- It depends on the quality of the net whether it will be admitted in the existing population. A new net will be accepted if its performance is better as the worst net in the pool. Then the worst net is replaced by the new one.

This topology process is steadily repeated. It can only be terminated by the user.

5.2 Results

XOR–Problem

To test our genetic algorithm, we tried to generate an optimal net for the XOR problem. 10 different nets were made by chance and the evolution server were started on several computers (6 x Sun4, 4 x NeXT). The number of the training epochs per net was fixed on 1000 passes. After a short time (approximately 5 minutes) the genetic algorithm converged to the well known topology for the XOR Problem (Figure 6).

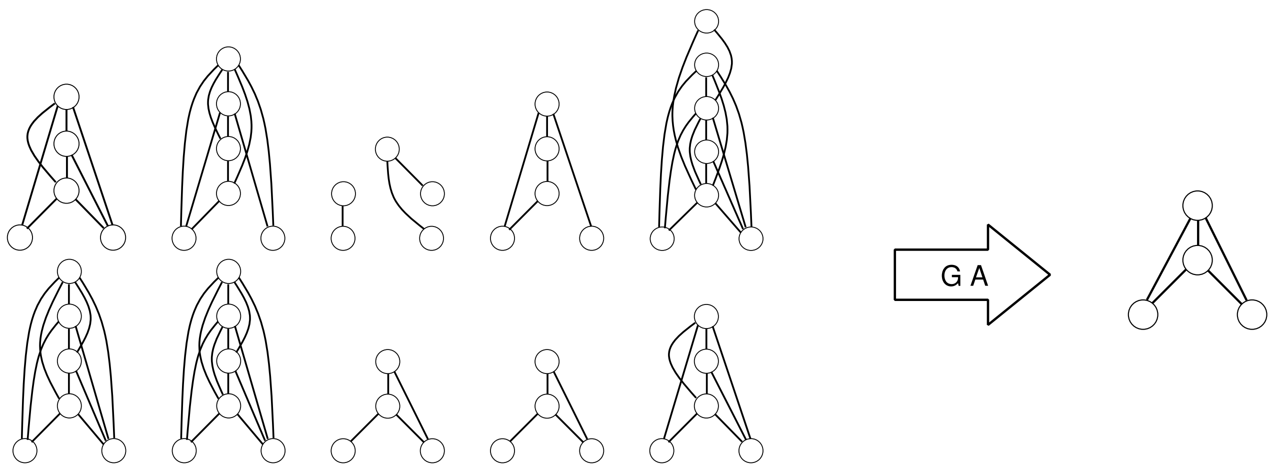

Figure 6: The starting nets for the XOR Problem. All nets were made by chance. The genetic algorithm converged after approximately 5 minutes to the minimal topology for the XOR Problem.

The development of the net size during the evolution is demonstrated in Figure 7. It shows the number of units and connections of the nets (average values of the population). It can be seen that at first big topologies appear more frequently in the population. This is because they achieve better results at the beginning of the evolution. Later, however, the algorithm preferred smaller architectures.

Thyroid–data

The thyroid data are measurements of the thyroid gland. Each measurement consists of 21 values — 15 binary and 6 analog. Three classes are assigned to these values which correspond to the hyper–, hypo–, and normal function of the thyroid gland. Since over 92% of all patients have a normal function, a useful classifier must be significantly better than 92% correct classifications. The training set consists of 3772 measurements and again 3428 measurements are available for the testing set. This classification problem is hardly to solve for neural nets. Several fixed topologies were tested. In order to maintain a continuous text flow all figures have been placed at the end of this section.

Figure 8 shows the course of the error during the training of these fixed topologies. It can be seen that several thousand learning passes are necessary to archive a reasonable classifier. To get an optimal learning rate and momentum we trained the 21x10x3 net with different values. Once the learning rate was fixed at 0.01 and various momentum values were tested. Then the momentum was set to 0.9 to test different learning rates (Figure 9 and 10). It can be seen that small values slow down learning. On the other hand, oscillations arise when large values are assigned to the learning rule. Thus, we set the learning rate constantly to 0.01 and the momentum to 0.9. These values were found as a good compromise between learning speed and a smooth course of the error.

In order to accelerate the evolution process, we don't use the complete training set. Instead of that 713 measurements were chosen out of it. These selected patterns are the hardest ones of the training set because they are drawn from the boundary region between two categories. All nets were trained only with this extracted set. The training period during the evolution was fixed on 500 passes. Learning rate and momentum were adjusted constantly to 0.01 and 0.9 as well. To start the evolution process we designed 10 different nets by chance. These nets have various numbers of units and connections (table 3).

	#units	#Weights	Connectivity
n_1.net	54	110	9.0%
n_2.net	34	270	22.4%
n_3.net	49	461	47.9%
n_4.net	34	32	9.2%
n_5.net	39	378	71.6%
n_6.net	54	174	14.3%
n_7.net	24	40	63.5%
n_8.net	39	433	82.0%
n_9.net	31	233	92.5%
n_0.net	34	238	31.6%

Table 3: Characteristics of the starting nets for the thyroid task.

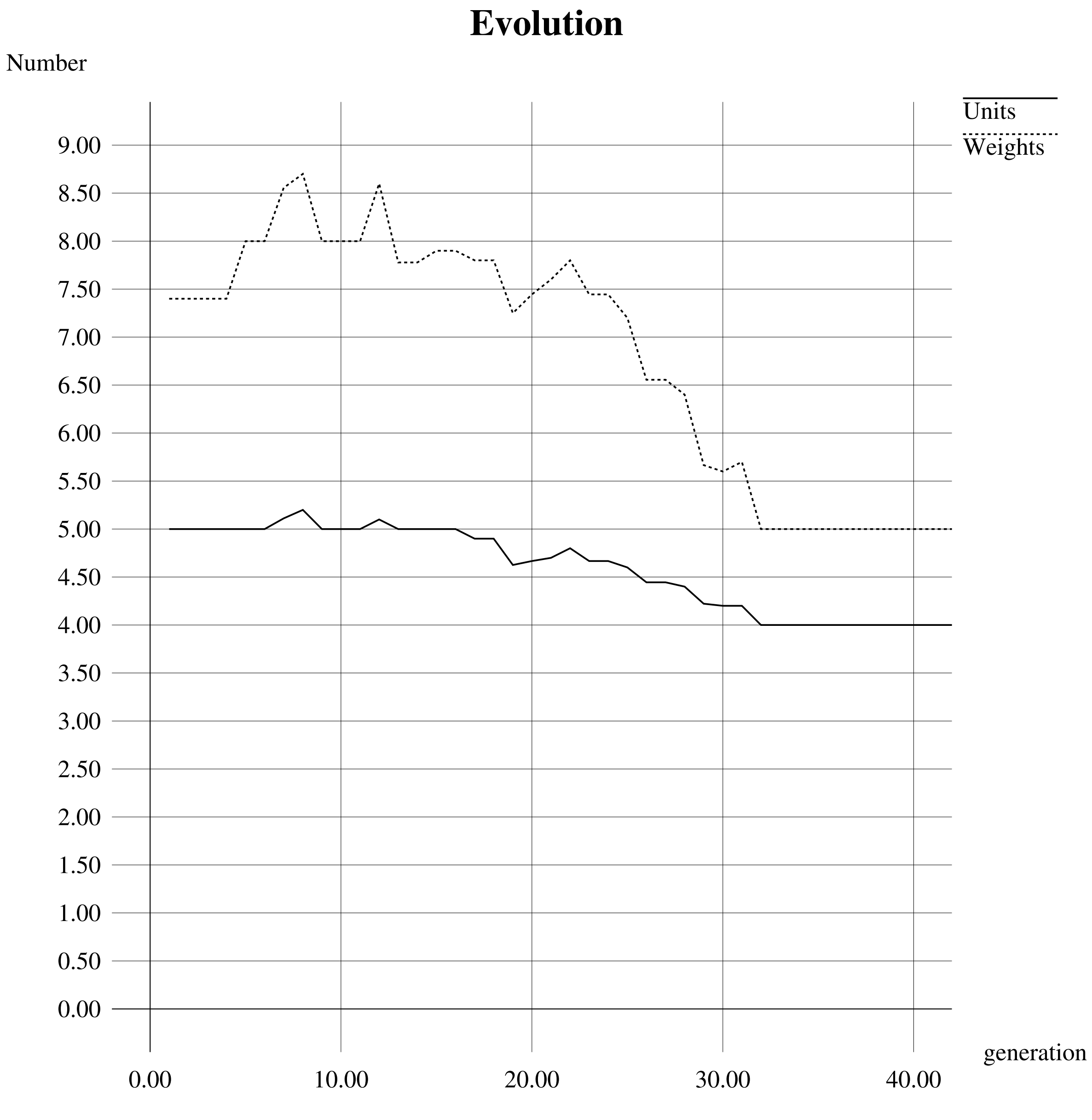

Figure 7: The development of the net size for the XOR problem during the evolution process. After approximately 30 generation (e.g. 30 developed nets) the architectures converged to the minimal topology which consists of 4 units and 5 connections.

The achieved classification performance during the evolution process is shown in Figure 11. Only the values of the best and the worst net of each generation are represented. To get these well adapted networks the evolution needs about 72 hours. It can be seen that the best net reaches a classification performance of more than 95% correct classified patterns.

The development of the net size is demonstrated in Figure 12. It can bee seen that the evolution process of the thyroid nets shows the same behaviour like the generated XOR nets. At first big topologies are preferred because of the better results. Later the algorithm produced smaller architectures.

Table 4 compares the results of the nets evolved by evolution with the results of the fixed architectures. To obtain comparable results, every fixed net was also trained 500 epochs with the extracted training set. The table represents the achieved classifications of the extracted training set, the whole training set, and the testing set. Note, that the nets achieve a better classification performance on the complete training set. This is, because the extracted set consists only of the hardest values of the whole training set. Furthermore, the table shows the number of units and connections in order to compare the sizes of nets. It should be taken into account that the number of the connections is especially decisive for the size of a net. The adapted nets are more efficient although they are smaller. They train faster and generalize better. To reach a similar classification performance with the fixed nets, many additional learning passes would be necessary.

	#Units	#Weights	Connectivity	used training set	whole training set	test set
fixed nets						
21x5x3.net	29	183	94.8%	76.6%	95.0%	94.3%
21x10x3.net	34	303	87.1%	76.9%	95.5%	94.8%
21x20x3.net	44	543	74.1%	71.1%	94.4%	93.9%
evol. nets						
n_123.net	48	285	31.1%	94.8%	98.3%	97.3%
n_125.net	49	283	29.4%	95.0%	98.1%	96.9%
n_130.net	49	278	28.9%	94.8%	98.3%	97.4%
n_136.net	48	277	30.3%	94.8%	98.2%	97.1%
n_147.net	49	276	28.7%	95.0%	98.4%	97.3%
n_149.net	50	279	27.6%	95.0%	98.1%	97.2%
n_152.net	50	278	27.5%	95.0%	98.2%	97.0%
n_154.net	49	286	29.7%	95.1%	98.3%	97.3%
n_155.net	50	279	27.6%	95.0%	98.6%	97.4%
n_156.net	50	278	27.5%	94.8%	98.4%	97.5%

Table 4: Comparison between the fixed and the generated networks. The evolved architectures archive a better performance although they are smaller.

Finally we investigated the training behavior of the generated nets shown on the complete training set. Figures 13 and 14 show the error of selected nets during the learning passes. The illustrations demonstrate that the error starts to oscillate. This behavior is typical for all generated nets. Nevertheless, the error of the adapted nets is below the error of the fixed nets.

	training set	test set
21x5x3.net	99.0%	97.4%
21x10x3.net	99.1%	97.3%
21x20x3.net	99.3%	97.4%
n_123.net	99.5%	98.2%
n_125.net	99.4%	98.6%
n_130.net	99.4%	98.4%
n_147.net	99.4%	98.2%
n_152.net	99.4%	98.5%
n_154.net	99.4%	98.4%
n_155.net	99.4%	98.6%
n_156.net	99.4%	98.4%

Table 5: Comparison between the fixed and the generated networks which are trained with the whole training set.

Table 5 compares the results of the trained nets. It shows that the adapted nets have a some better result concerning the training set. But with respect to the generalization behavior they are, however, superior to the fixed architectures ($> 1\%$). This can be explained by the fact that the evolution finds feature extracting cells.

Figures

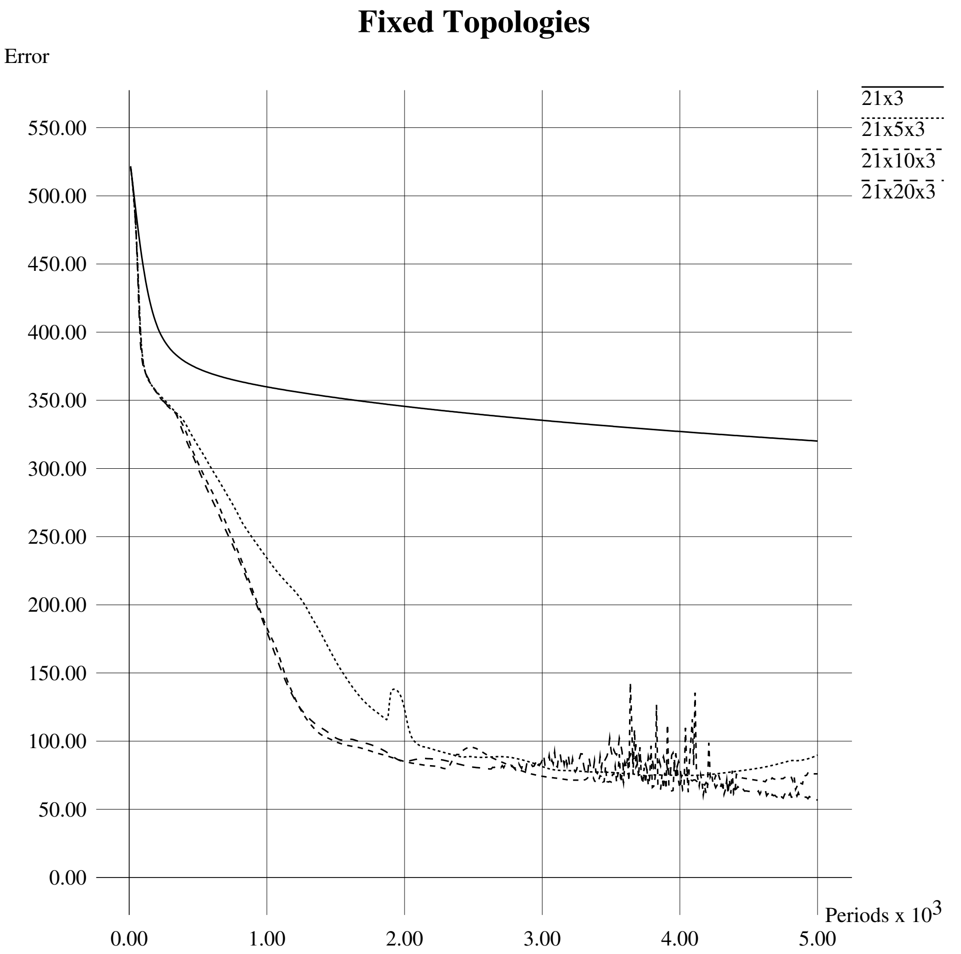

Figure 8: The error during the training of the thyroid data with fixed topologies. The name of a net specifies its architecture. For example the name 21x10x3 indicates that the net consists of 21 input units, 3 output units and 10 hidden units. All units are fully interconnected. Learning rate and momentum were set to 0.01 and 0.9

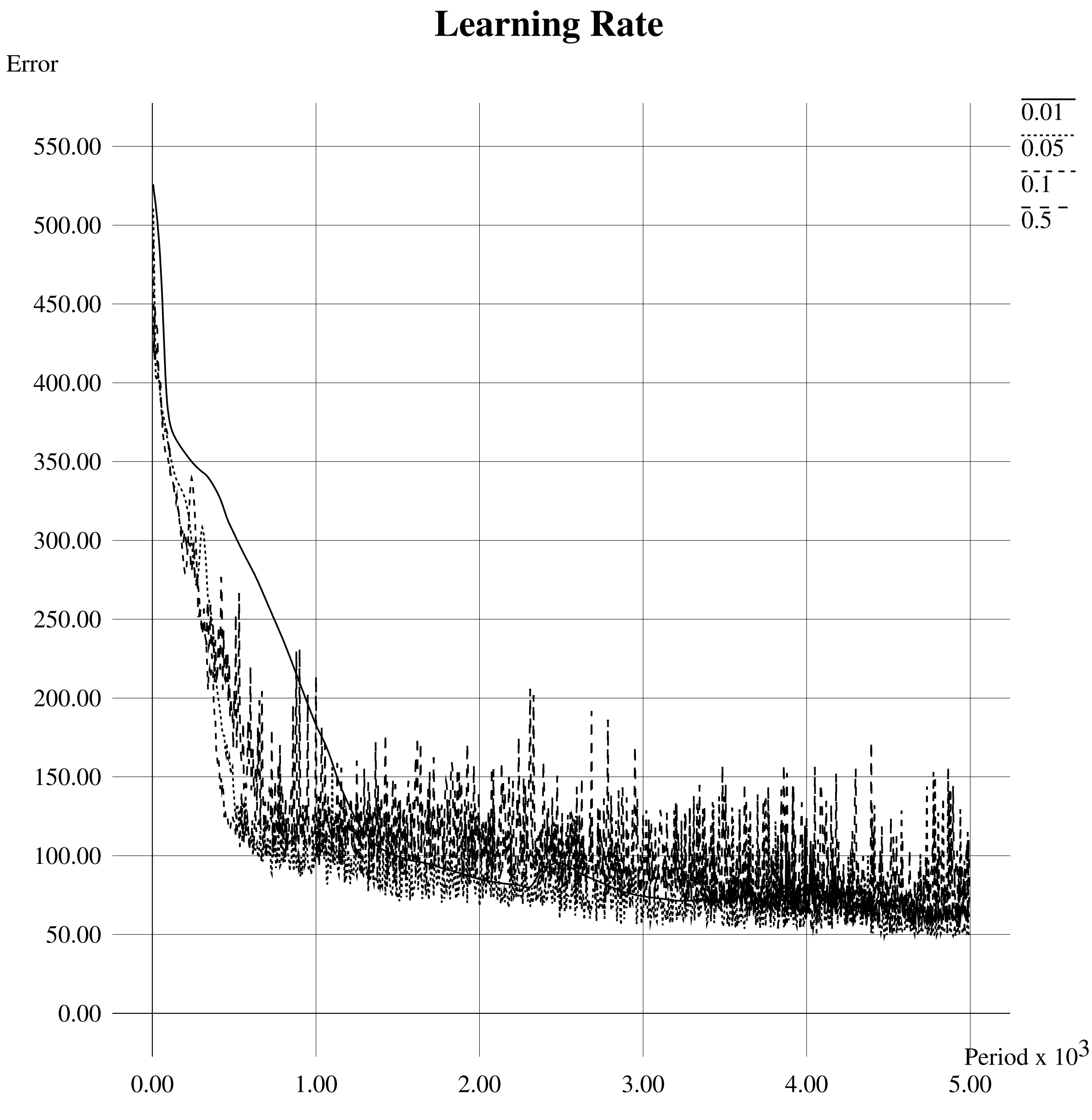

Figure 9: The error during training the 21x10x3 net. In order to test various learning rate values the momentum term was set to 0.9

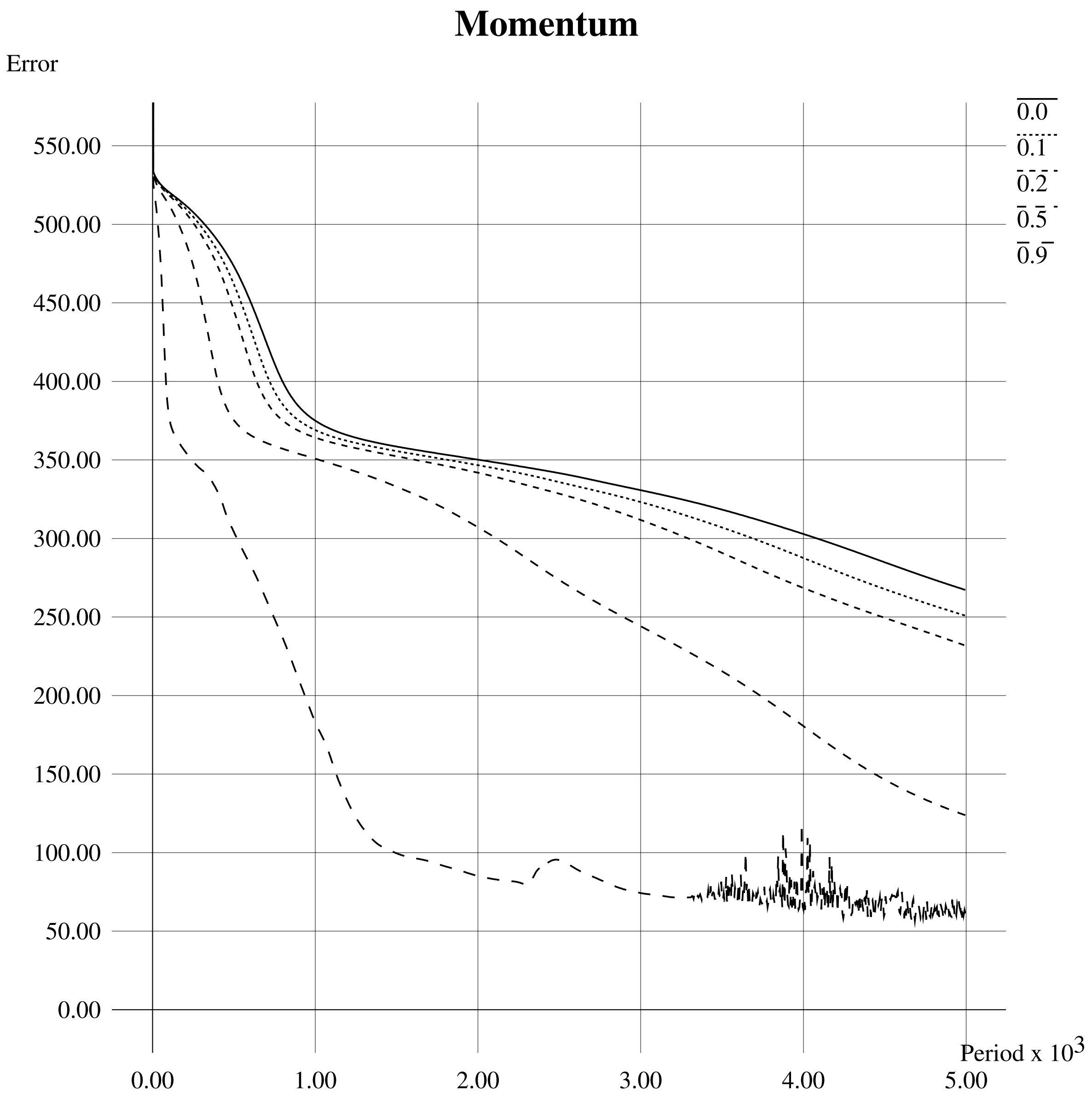

Figure 10: The course of error produced by various momentum values. The learning rate was fixed at 0.1

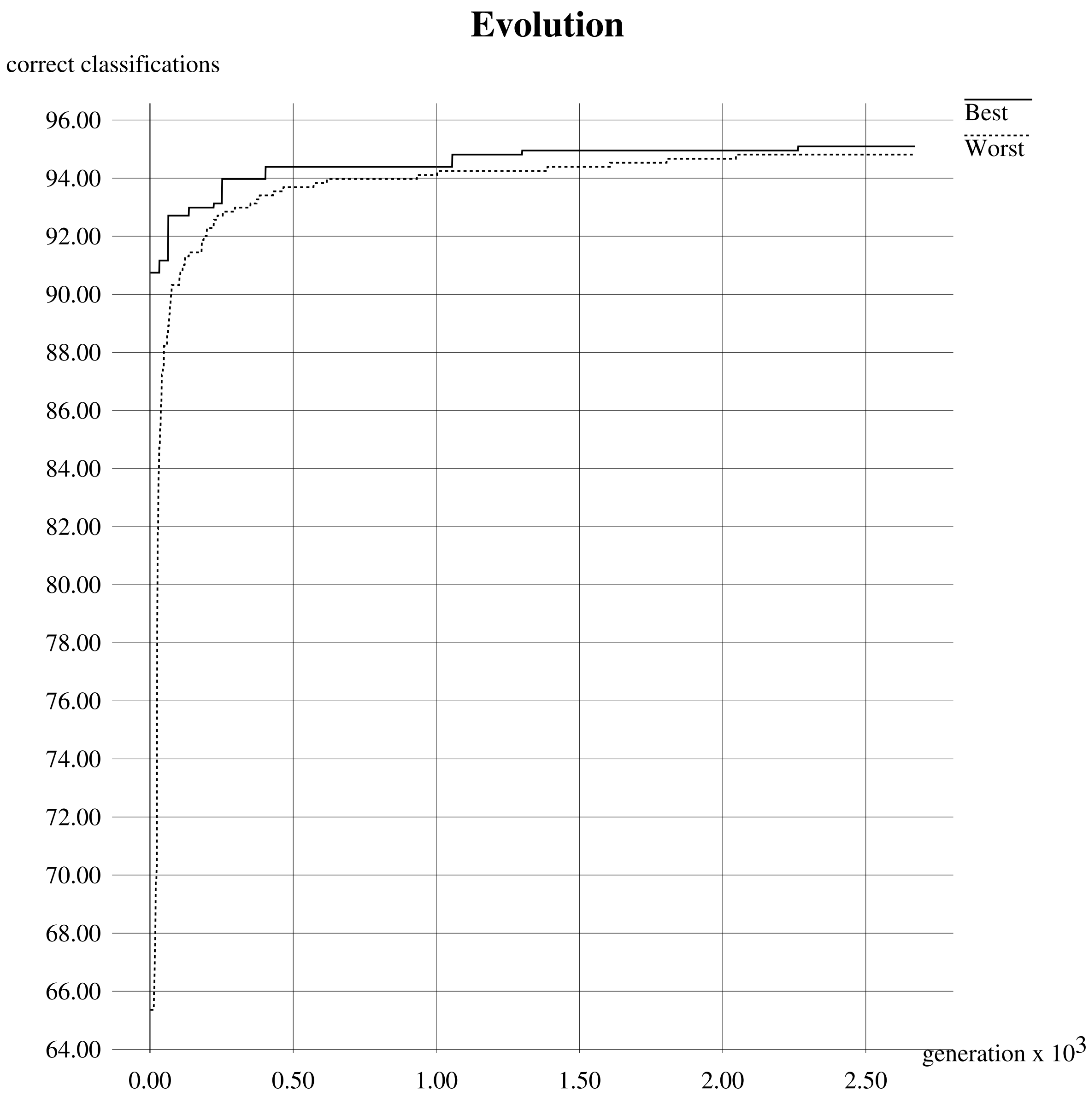

Figure 11: The achieved classification performance during the evolution process. Only the classification performance of the best and the worst net of the population is shown.

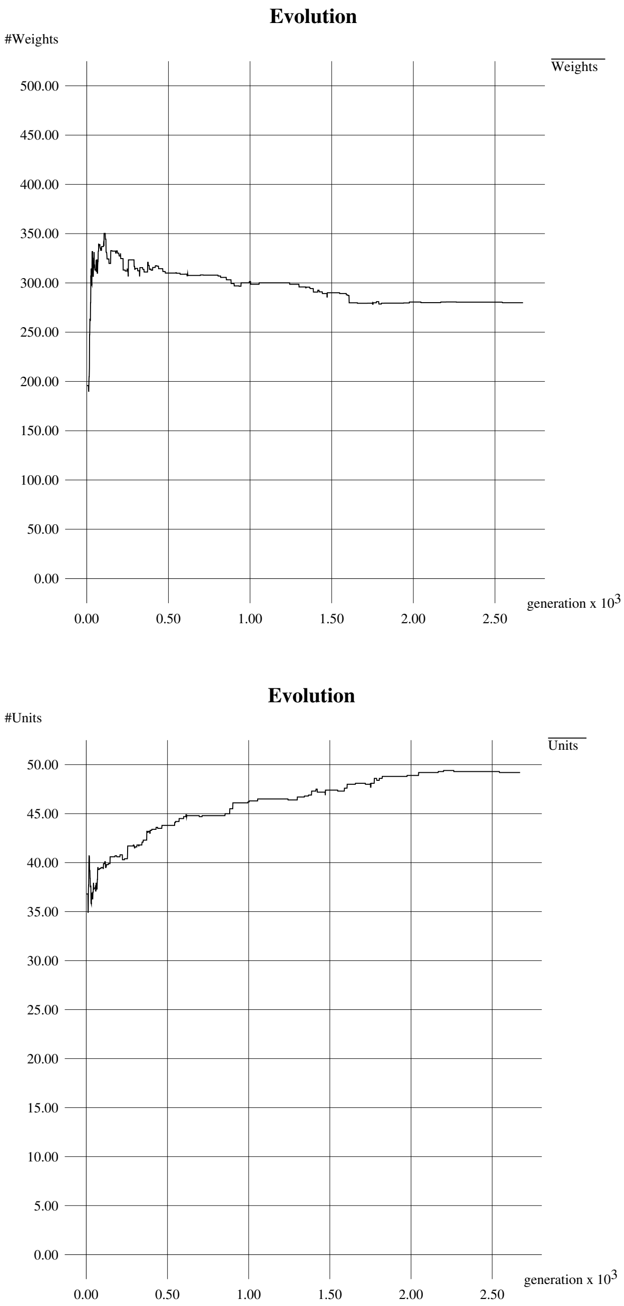

Figure 12: The development of the net size during the evolution process.

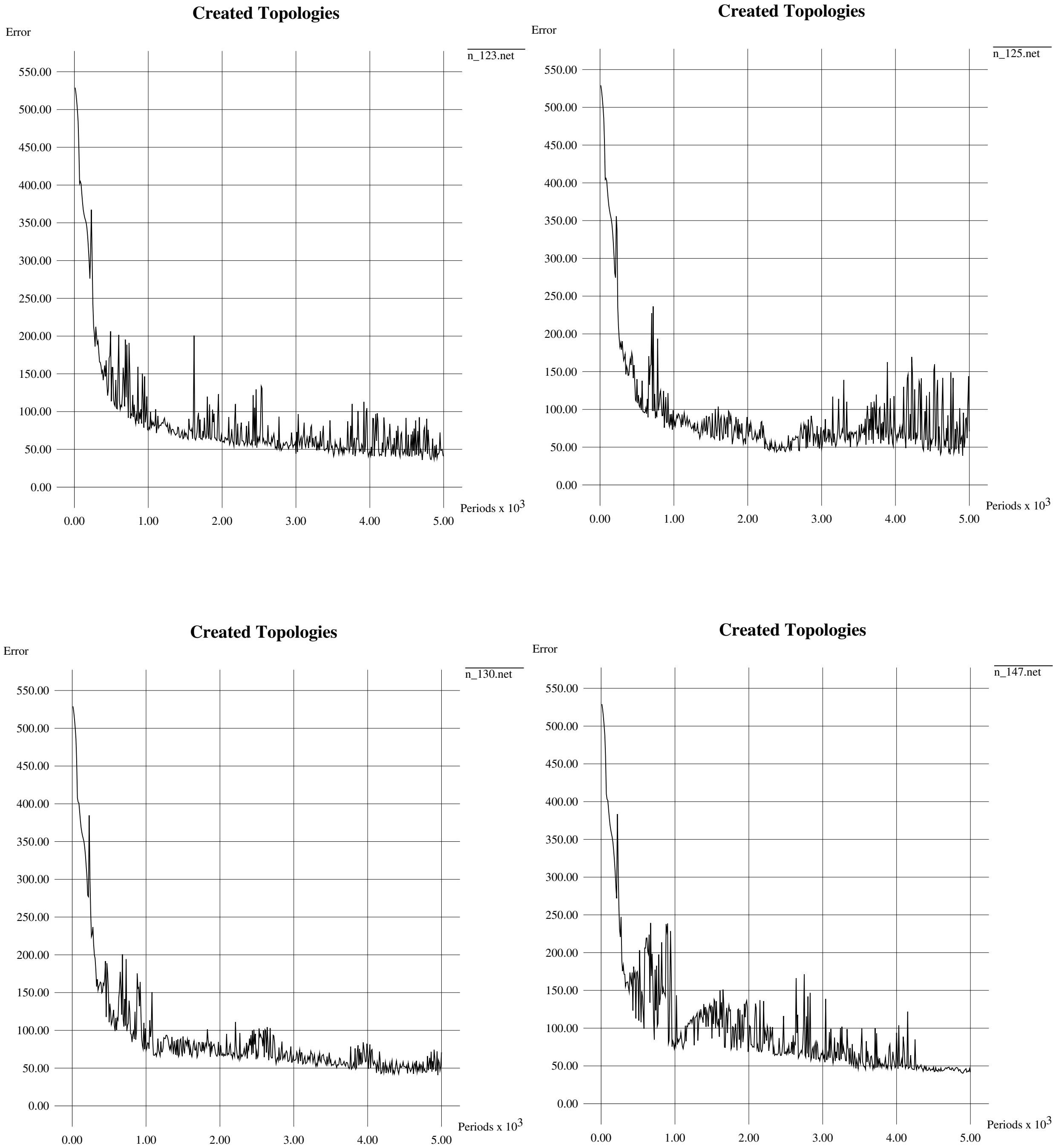

Figure 13: Error during the training of the evolved networks by the complete thyroid data set

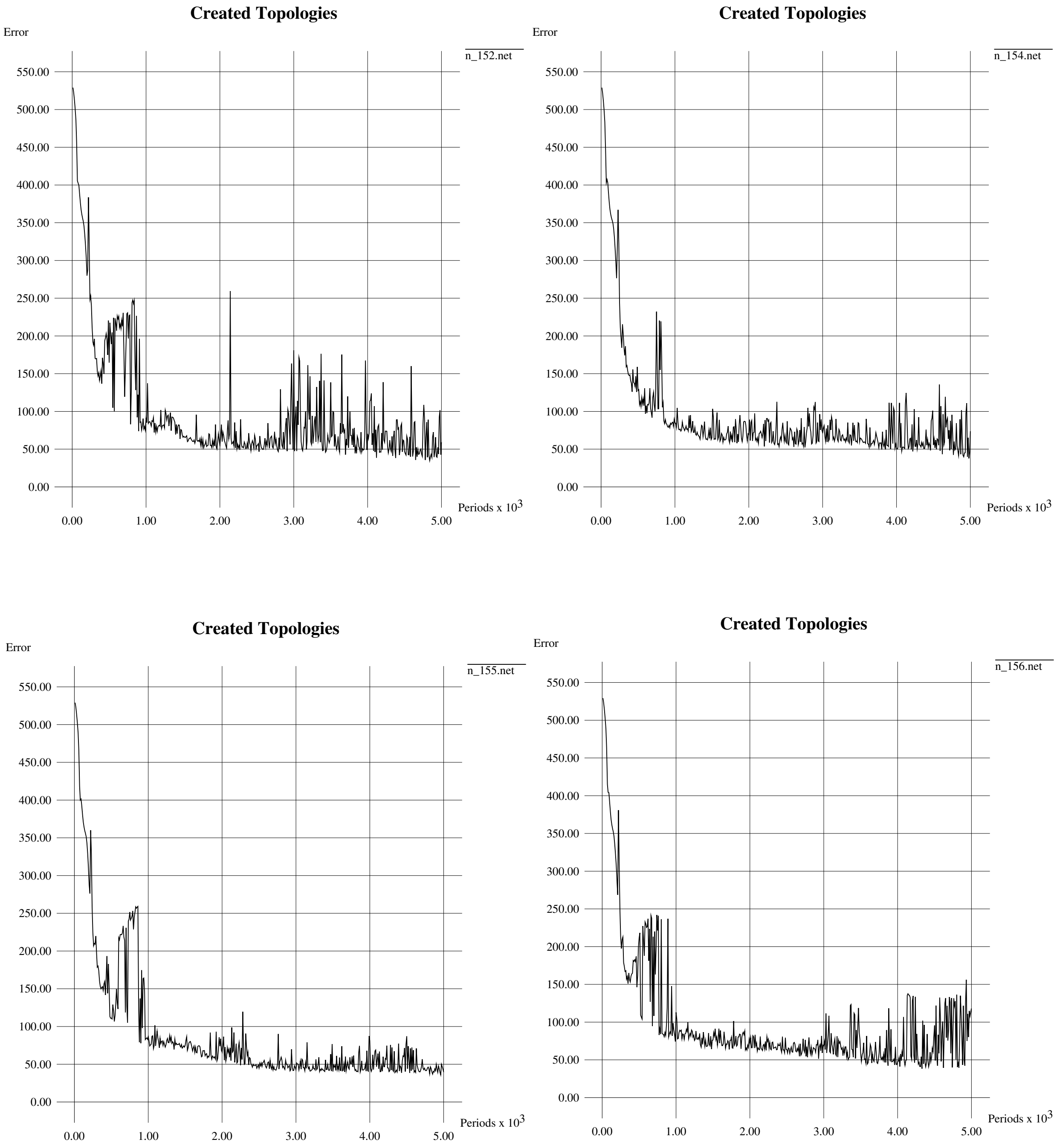

Figure 14: Error during the training of the evolved networks by the complete thyroid data set (continued)

6 Conclusion and further work

Unfortunately, the generated nets show oscillations in the course of error. These oscillations start typically after the number of epochs which was used for the genetic algorithm (here at 500 epochs). The genetic algorithm produced nets which can be trained quickly up to this epoch. The training behavior of the nets after this number of epoch is uncertain. In order to suppress these oscillations, a learning rate adaption would seem to be useful.

Furthermore we have discovered that the number of training epochs can be reduced by using quickprop [Fahlman, 1988]. We will also test this training procedure in our algorithm. In order to improve the generalization behavior of the nets, the performance of the produced net should not be determined with the utilized training set. It would be better to check the quality through an second pattern set. The real generalization behavior must then be determined on a third pattern set. Concerning the example of the thyroid data the nets could be trained with the extracted training set, but the quality measure should be determined with the complete training set. The test set could, in this case, be used as the generalization test.

7 Acknowledgements

This work is supported by the *Deutsche Forschungsgemeinschaft* (DFG) as part of the project *FE–generator* (grant Schi 304/1–1).

8 References

The references are sorted historically.

Minsky and Papert, 1969 : *Perceptrons*, MIT Press

Rechenberg I., 1973 : *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommberg–Holzboog, Stuttgart

Schwefel H.P., 1976 : *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie*, Birkhäuser, Basel, Stuttgart

Mjolsness E. and Sharp D.H., 1986 : *A Prelimiary Analysis of Recursively Generated Networks*, in Denker J. (Eds.): *Neural Networks for Computing*, Snowbird (Utah)

Rumelhart D.E., Hinton G.E. and Williams R.J., 1986 : *Learning internal representations by error propagation*, in Parallel Distributed Processing: *Explorations in the Microstructures of Cognition*, Vol.I, MIT Press, pp. 318–362

Chan L.W. and Fallside F., 1987 : *An Adaptive Training Algorithm for Backpropagation Networks*, Computer Speech and Language, Cambridge (MA), pp. 205–218

Lehar S. and Weaver J., 1987 : *A Developmental Approach to Neural Network Design*, Proc. of the IEEE Intern. Conf. on Neural Networks, Vol. I, pp. 97–104

Longstaff I.D. and Cross J.F., 1987 : *A pattern recognition approach to understanding the multilayer perceptron*, Pattern Recognition Letters 5, pp. 315–319, Elsevier

Mjolsness E., Sharp D.H. and Alpert B.K., 1987 : *Recursively Generated Neural Networks*, Proc. of the IEEE Intern. Conf. on Neural Networks, Vol. III, pp. 165–171

Hubel D.H., 1988 : *Eye, Brain and Vision*, Scientific American Library

Jacobs R.A., 1988 : *Increased Rates of Convergence Through Learning Rate Adaptation*, Neural Networks, Vol. I, Pergamon Press, pp. 295–307

Whitley D., 1988 : *Applying Genetic Algorithms to Neural Net Problems*, Neural Networks, Vol. 1, p. 230

Hornik K., Stinchcombe M. and White H., 1989 : *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, Vol. 2, pp. 359–366, Pergamon Press

Whitley D., 1988 : *Applying Genetic Algorithms to Neural Network Learning*, Technical Report, Department of Computer Science, Fort Collins (Colorado), pp. 137–144

Goldberg D., 1989 : *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison–Wesley

Harp S.A., Samad T. and Guha A., 1989 : *Towards the Genetic Synthesis of Neural Networks*, Proc. of the third Intern. Conf. on Genetic Algorithms (ICGA), pp. 360–369, San Mateo (CA)

Harp S.A., Samad T. and Guha A., 1989 : *The Genetic Synthesis of Neural Networks*, Technical Report CSDD–89–14852–2, Honeywell, Golden Valley (MN)

Hecht–Nielsen R., 1989 : *Theory of Backpropagation Neural Networks*, Proc. of the Intern. Joint Conf. on Neural Networks, Washington D.C.

Heistermann J., 1989 : *Parallel Algorithms for Learning in Neural Networks with Evolution Strategy*, Parallel Computing, Vol. 12

Miller G., Todd P.M. and Hegde S.U., 1989 : *Designing Neural Networks Using Genetic Algorithms*,Proc. of the third Intern. Conf. on Genetic Algorithms (ICGA), pp. 379–384, San Mateo (CA)

Mühlenbein H. and Kindermann J., 1989 : *The Dynamics of Evolution and Learning — Towards Genetic Neural Networks*, in Pfeifer et al. (Eds.): *Connectionism in Perspective*, Elsevier

Montana D. and Davis C., 1989 : *Training Feedforward Neuronal Networks Using Genetic Algorithms*, Technical Report, BBN Systems and Technologies Inc., Cambridge (MA)

Schiffmann W.H., 1989 : *Selbstorganisation neuronaler Netze nach den Prinzipien der Evolution*, Fachbericht Physik Nr. 7, Universität Koblenz

Whitley D. and Hanson T., 1989 : *Optimizing Neural Networks Using Faster, More Accurate Genetic Search*, Proc. of the third Intern. Conf. on Genetic Algorithms (ICGA), pp. 391–396, San Mateo (CA)

Whitley D., 1989 : *The GENITOR Algorithm and Selection Pressure: Why Rank–Based Allocation of Reproductive Trials is Best*, Proc. of the third Intern. Conf. on Genetic Algorithms (ICGA), pp. 116–121, San Mateo (CA)

de Garis H., 1990 : *Genetic Programming — Modular Neural Evolution for Darwin Machines*, Proc. of the Intern. Joint Conf. on Neural Networks, Vol I, pp. 194–197

Dodd N., 1990 : *Optimization of Network Structure Using Genetic Techniques*, Proc. of the Intern. Conf. on Neural Networks, Paris

Kitano H., 1990 : *Empirical Studies on the Speed of Convergence of Neural Network Training Using Genetic Algorithms*, Proc. of the National Conf. of th American Association of Artificial Intelligence (AAAI), pp. 789–795

Heistermann J., 1990 : *Learning in Neural Nets by Genetic Algorithms*, Proc. of Parallel Processing in Neural Systems and Computers (ICNC), Eckmiller R. et al. (Eds.), pp. 165–168, Elsevier

Schiffmann W.H. and Mecklenburg K., 1990 : *Genetic Generation of Backpropagation Trained Neural Networks*, Proc. of Parallel Processing in Neural Systems and Computers (ICNC), Eckmiller R. et al. (Eds.), pp. 205–208, Elsevier

Silva F.M. and Almeida L.B., 1990 : *Speeding up Backpropagation*, Advanced Neural Computers, pp. 151–158 Elsevier

Weiss, G. : *Combining neural and evolutionary learning: Aspects and approaches*, Report FKI–132–90, Technische Universität München, 1990

Whitley D. and Bogart C., 1990 : *The Evolution of Connectivity: Pruning Neural Networks Using Genetic Algorithms*, Proc. of the intern. Joint Conf. on Neural Networks, Vol. I, pp. 134–137

Whitley D., Starkweather and Bogart C., 1990 : *Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity*, Parallel Computing Vol. 14, pp. 347–361, Elsevier

Schiffmann W.H., Joost M. and Werner R., 1991 : *Performance Evaluation of Evolutionarily Created Neural Network Topologies*, Proc. of Parallel Problem Solving from Nature, Schwefel H.P. and Maenner R. (Eds.), pp. 274–283, Lect. Notes in Computer Science, Springer

Merrill J. and Port R., 1991 : *Fractally Configured Neural Networks*, Neural Networks, Vol. 4, pp.53–60

Mühlenbein H., 1992 : *Parallel Genetic Algorithms and Neural Networks as Learning Machines*, Parallel Computing '91, Evans et al. (Eds.), Elsevier

Mühlenbein H., 1992 : *Parallel Genetic Algorithms in Combinatorial Optimization*, Proc. ORSA, Osman Balci (Ed.), Pergamon Press

Mühlenbein H., 1992 : *Asynchronous Parallel Search by the Parallel Genetic Algorithm*, 3rd. IEEE Symposium on Parallel and Distributed Processing, pp. 526–533

Schiffmann W.H., Joost M. and Werner R., 1992 : *Optimierung des Backpropagation Algorithmus zum Training von Multilayer Perceptrons*, Fachbericht Physik, 15/1992, Universität Koblenz