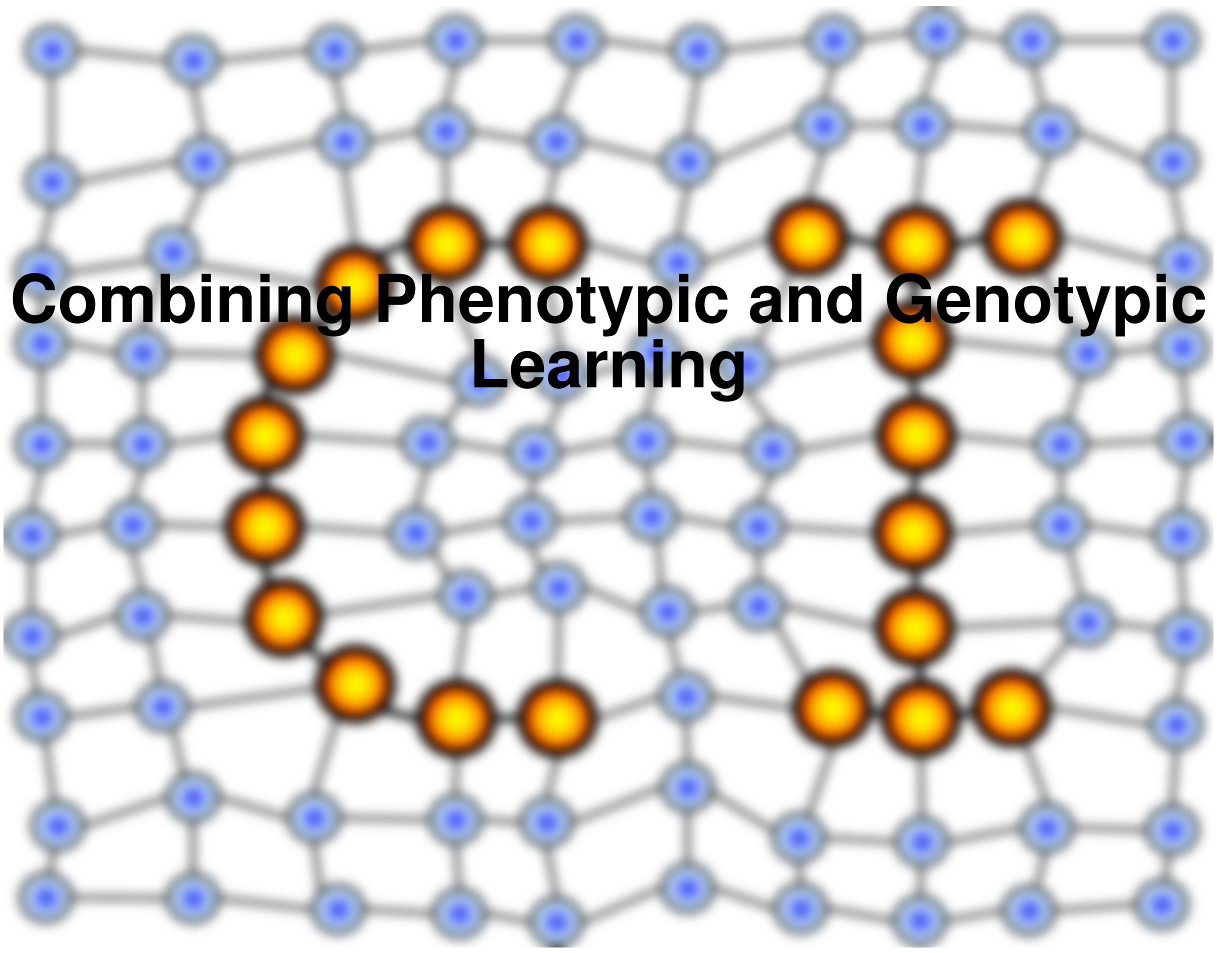
MACHINE LEARNING REPORTS



Combining Phenotypic and Genotypic Learning

Report 01/2008
Submitted: 15.04.2008
Published: 18.04.2008

Udo Seiffert
Scottish Crop Research Institute (SCRI)
Mathematical Biology
Invergowrie, Dundee, UK
and
University of Magdeburg
Inst. for Electronics, Signal Processing, and Communications (IESK)
Neural Systems
Magdeburg, Germany

Abstract

The terms *phenotypic* and *genotypic learning* refer to naturally inspired adaptive algorithms, based on the information processing in the brain and evolution of individuals of a population, respectively. Their appearance might be, among others, *artificial neural networks* and *genetic algorithms*, both fundamental columns of *computational intelligence*, but serving rather different purposes. There are many approaches to combine both worlds, particularly to optimize the many control parameters of neural network design and training using genetic algorithms as optimization tool. Beyond that, this paper demonstrates the *direct* combination of the underlying principles of phenotypic and genotypic learning to obtain neural network training that is qualitatively improved compared to the traditional method to apply genetic algorithms to neural network training. This is achieved by making the GA aware of the semantic that is coded in its chromosomes. The results clearly show the advantages of this method.

1 Introduction

The recently increasing research area of computational intelligence is considered to mainly consist of three major techniques - artificial neural networks, genetic algorithms, and fuzzy logic. All three components have in common to be inspired by nature and have been subject to a steadily growing interest. The research activities have been rapidly forced throughout the last years. This has paved the way to develop some hybrid techniques such as a combination of neural networks and genetic algorithms as considered in this paper. Artificial neural networks (ANN) are concerned with learning of individuals, often called *phenotypic* learning, whereas genetic algorithms (GA) deal with the adaptation of a population to a changing environment, so called *genotypic* learning. Evolution in a wider sense builds the base for this class of optimization methods.

Genetic or evolutionary algorithms were significantly inspired and developed by HOLLAND [1] in the 1970s. They are based on the Darwinian theory of the *survival of the fittest*. This strategy, whereby potential solutions (individuals) to a problem compete and mate with each other in order to get increasingly better individuals (by means of their genotype), offers an efficient search method for a complex problem space that is mapped onto a corresponding gene space. Each solution to a problem to be solved is represented by a single individual and its genotype, respectively [2].

Generally, neural networks offer an attractive paradigm to solve many real-world problems, e.g. pattern classification, clustering, function approximation, associative memory, non-linear system modelling, control as well as prediction and forecasting. They are robust in the presence of noise, fault tolerant, and suitable for massively parallel computation [3, 4]. Due to their important feature of learning by example, neural nets are often used in applications with little or incomplete understanding of the problem to be solved but available training data [5]. In contrast to genetic algorithms, neural networks tackle a particular problem within the original data space – the phenotype.

Practical implementations of ANNs require the choice of a suitable network topology and a proper learning strategy as well. Since the performance of an ANN strongly depends on the used network architecture, its design is very important. Therefore, some investigations [6, 7, 8, 9] have been done to use GAs to search the space of potential ANN architectures for optimal or at least sub-optimal but satisfactory designs. That concerns above all the network type and its topology. This is one possible intersection between neural nets and genetic algorithms. Due to particular features of different network types, practical implementations are often depending on the target system [10, 11, 12]. In the end, this automates only the network design and not its training.

Once a suitable network type and topology has been found, this network must be trained by an appropriate learning scheme. A number of free parameters must be handled again. In [13, 14, 15, 16] some suggestions were made to integrate at least some of these parameters into the genetic algorithm and to combine the above mentioned design optimization with the network training. This leads to some important advantages. The design and training processes become one self-contained step. The user does not have to care about possibly confusing and complex design and learning parameters. However, the black-box character of neural nets becomes even more dominant.

Nevertheless, this process remains very difficult (handling the parameters of the GA), yielding only limited results on rather selective problems. In many of these approaches genetic algorithms have been used to optimize global network parameters. The core network control and training strategy has been changed scarcely. In other words, these approaches usually lead rather to more comfort for the user than to a better performance of the neural network.

Another possibility to join ANNs and GAs in a natural way is to enhance parts of the network control and training strategy by an evolutionary algorithm. Independent of how a network was designed, either conventionally or by a genetic algorithm, a specific component of the training scheme, to be motivated in the next section, can be improved or even substituted. Due to the global and universally valid character of this strategy, this network is not limited to any particular problem. It can be used and handled the same way as the original network, but may lead to better results. Moreover, the user retains more control of the network's learning and is able to tune its performance manually.

These considerations define the scope of this paper: improving the *performance* of *phenotypic* learning, as key step in neural network development, by combining it with the advantages of *genotypic* learning in terms of the natural principles of evolution.

2 Motivation

The very frequently used Multiple Layer Perceptron (MLP) [17] has its roots in the simple Perceptron introduced by ROSENBLATT [18], which was able to classify only linearly separable input patterns. Due to its multiple layer structure, MLPs can separate complex non-linearly separable classes. An error value is calculated based on a comparison between desired and actual output response of the network. PARKER [19] and RUMELHART [20] are associated with *Backpropagation*, the commonly used MLP control and training scheme. It got its name from how it handles these error values and stands even as a synonym for this network type by now. Because it is wide-spread and really a standard algorithm, the MLP is not explained in detail now, but only as it is necessary to understand the further paper. For more information refer for instance to [21].

A key assumption of Backpropagation is that all neurons, resp. all weights, are somewhat responsible for an occurring output error. The global error value Q of training example s is a function of all weights **w**, the actual output **y** and desired output **d** as well as the input vectors **x**.

$$Q^{(s)} = Q^{(s)}(w_1 \dots w_l, y_1 \dots y_m, d_1 \dots d_m, x_1 \dots x_n) \tag{1}$$

Responsibility for the global error of the network is affixed by propagating the output response error backward through all weighted connections to the previous layers until the input layer is reached. This assumes a differentiable and continuous transfer function of all neurons. Commonly the *sigmoid* or *hyperbolic tangent* functions are applied.

The aim of the training process is to minimize the global error

$$Q = \sum_{(s)} Q(s) \rightarrow \min \tag{2}$$

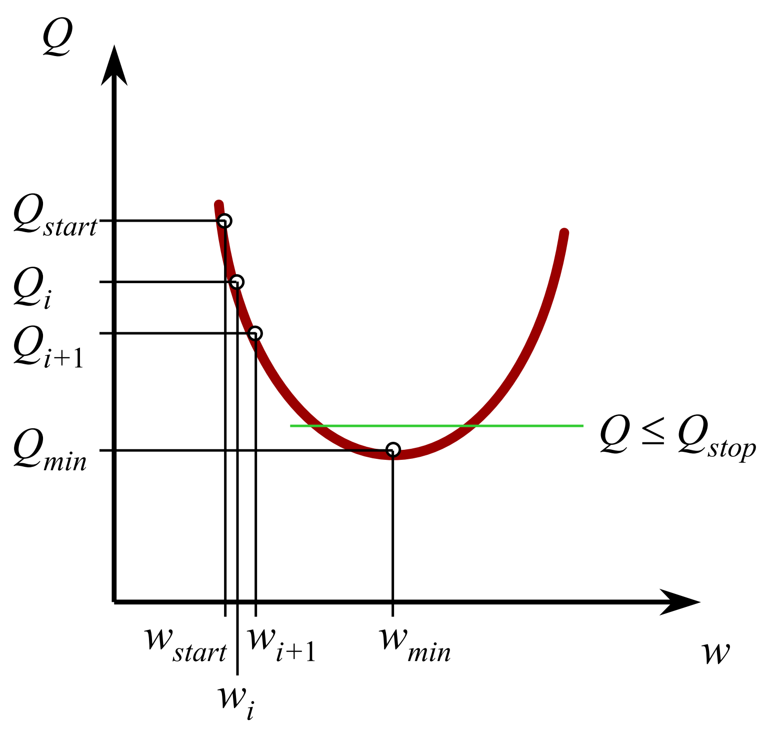


Figure 1: Gradient descent by means of a simple one-dimensional error surface. Beginning at an initial point (marked with index *start*), the aim of the learning process is to minimize the global error Q (indexed with i as it proceeds) until it falls below a given threshold Q_{stop}.

by modifying the weights **w**. For the update of a single weight element can be written

$$w_{j_{new}} = w_j - \gamma \cdot \frac{\partial Q}{\partial w_j} \tag{3}$$

where γ is the learning coefficient. Depending on the number of weights, the error spans a multi-dimensional error surface, which is more or less rugged. In other words, each weight will be changed according to the size and direction of a negative gradient on the error surface. Commonly this algorithm is called *gradient descent* (see Fig. 1).

Despite its popularity and a large number of significant improvements in the past, the gradient descent, as minimization (optimization) function for Backpropagation, has several drawbacks. It is dependent on the error surface, the initial weights, and some further parameters, (e.g., the learning coefficient γ). A common error surface has many local minima.

The system often suffers from getting stuck in a local minimum depending on the shape of the error surface and the size of the learning coefficient (see Fig. 2). If a plateau is entered, the size if the gradient (Eq. 3) is very small and its direction may alternate from one training step to the next. This is due to the local character of the gradient descent. Generally, global optimization schemes will ease these problems [22, 23].

Changing the weights as a linear function of the partial derivative (Eq. 3) assumes

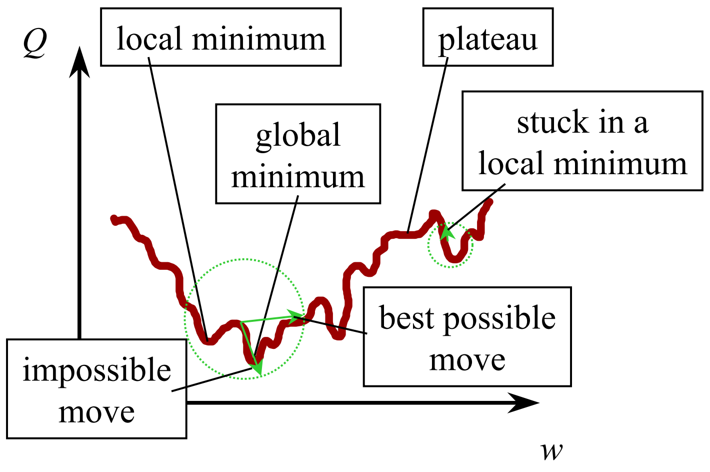


Figure 2: Stylized one-dimensional error surface showing the most significant problems of the gradient descent algorithm – local minimum, plateau, and the influence of the learning coefficient.

a piecewise (according to the size of the learning coefficient) locally linear error surface. Especially at points of high curvature this may lead to divergence. Although the concept of the momentum term (see [21] for a discussion) solves this particular problem, convergence of the gradient descent is still depending on the size of γ. If it is too small, the system suffers from very slow learning. On the other hand, a too large γ may prevent the system from finding a minimum. A synchronized decreasing learning coefficient is typically used.

Another assumption, as already mentioned above, the demand for a differentiable and continuous transfer function, leads sometimes to conflicts with other requirements, e.g. using alternative (non-differentiable) transfer functions to achieve a particular behaviour of the neurons.

Besides these disadvantages of the gradient descent algorithm, evolutionary algorithms are claimed to have a good performance for complex optimization problems to find near-optimal solutions. Thus, GAs promise to be a suitable substitution for several elements of a neural network control and training scheme. In the present case the complete Backpropagation algorithm will be replaced by a GA.

So far, the motivation to use GAs to substitute the gradient descent algorithm and error Backpropagation when training MLPs is mostly based on:

- drawbacks of the gradient descent algorithm,
- the demand for a differentiable and continuous transfer function in Backpropagation, and
- the suitability of genetic algorithms for large and complex optimization problems.

In order to allow a direct comparison between modified and original networks, in terms of both training results and computation time, a variation of the transfer function has been left out here.

3 Implementation

3.1 General Considerations

It would also go beyond the scope of this paper to explain the basic terminology, implementation, and properties of GAs. Therefore a key knowledge of genetic algorithms is assumed. Otherwise [2] may be a good introduction to GAs within this context. The effects of modified parameters controlling genetic algorithms will be explained in Sect. 4.

The first step is to develop an appropriate representation of the networks weights within the genotype. Here, it has to be distinguished between the *phenotype* or *problem space*, in other words the weight distribution over the neural network, and the *representation space*, the chromosomal structure of an individual within the GA, that is called *genotype*. Each potential solution to the problem is one weight set. Its size depends on the topology of the MLP. Each weight set has to be mapped onto the linear structure of one particular chromosome within the representation space as shown in Fig. 3.

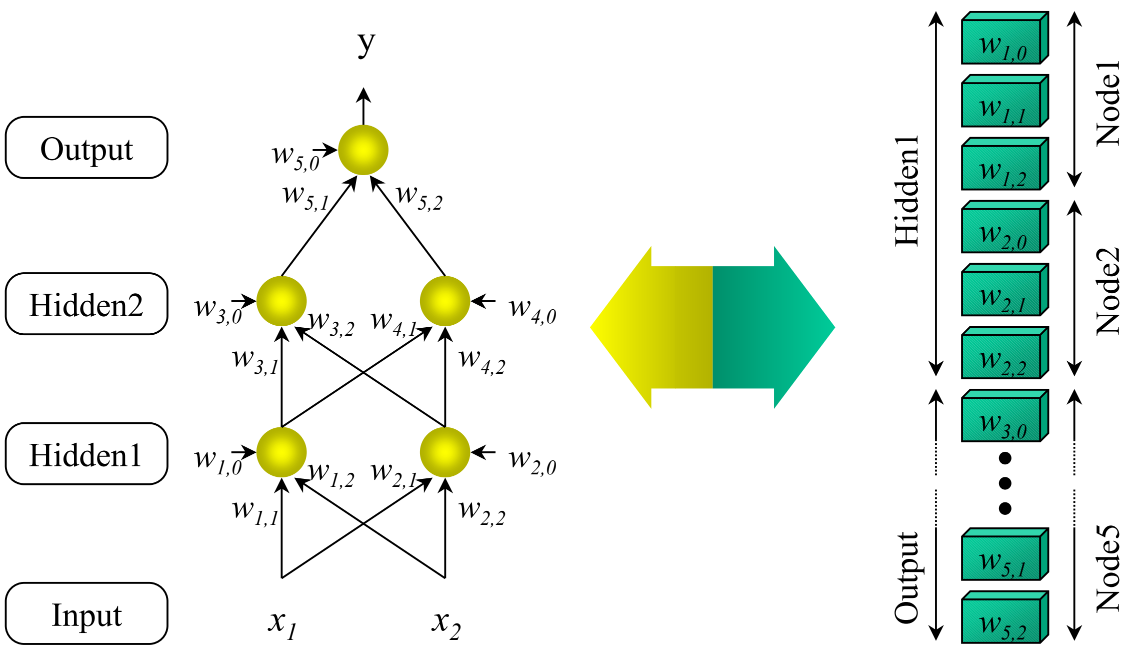


Figure 3: Mapping the weights of the neural network from *phenotype* or *problem space* (left side) onto a chromosome (*genotype; representation space*) and vice versa. Since the architecture of the ANN is predefined and remains fixed after the initialisation (see introductory section for detailed explanation), the chromosome solely consists of the weight values.

Being part of a population, each chromosome is subject to a sequence of genetic operations. In order to evaluate the fitness of a chromosome, it must be re-mapped into a weight set of the problem space, where just a recall of the ANN is performed. The choice of an appropriate and reversible chromosomal representation is rather important, because all further genetic operations are based on these chromosomes. A comprehensive discussion can be found in [7], although it mainly focuses on the evolutionary *design* of neural networks, and not their *training*. Since the network topology, including the transfer function of the neurons, is assumed to be predefined and remains fixed during the evolutionary *network training* process, it is not necessary to take any architectural information into consideration. Thus a chromosome solely consists of the network weights [24].

The initial population can be created now. This can be done by setting a given number of chromosomes to random values (*direct initialisation*). Uniform or normal distributions in a certain range are possible. If a more complex initialisation is required, e.g. in the context of the networks layered structure, the original initialisation procedure of the net must be run repeatedly until the desired number of the initial individuals is reached (*network initialisation*). Then each weight set has to be mapped onto its chromosomal representation.

Within the main loop (see Fig. 4) always the fitness of all members of the current population is evaluated at the beginning. The fitness is in the simplest case the mean error of all training examples during the network recall with one particular weight set being re-mapped from the chromosome.

The overall stopping criterion is based on the following single measures:

- desired network error,
- max. number of iterations (generations), and
- convergence to a single solution without changes from the previous to the current generation.

If one of these three criterions is met, the training is stopped and the fittest chromosome is re-mapped into the problem space and considered to be the final weight set of the satisfactorily trained ANN. Otherwise, a new generation has to be created. This task corresponds to the internal loop shown in Fig. 4 (center).

A fitness *normalization* is performed to scale the single fitness values of all chromosomes into a certain range. Another frequently applied algorithm, either combined with fitness normalization or run separately, is *ranking*. All chromosomes of the current population are ordered according to their fitness. These ranks are used by the selection mechanism instead of the direct fitness values. Ranking is used to force a constant selective pressure and ratio between the fittest and worst chromosome. Especially when several or even almost all fitness values are close together, either in the first few iterations after initialisation or near convergence, the fittest chromosome has always an equal lead on the second chromosome, and this again on the next, and so on.

After normalization and/or ranking is finished, a selection operator is applied. An intermediate population is created by selecting members from the current one. Several methods have been implemented and tested. The simplest operator is *uniform selection*, where, regardless of its particular fitness, each chromosome has an equal chance to be selected. Other, more complex methods are based on the previously computed fitness values. The most frequently applied operators are *integral selection* and *roulette wheel*, where the probability to be selected is proportional to the fitness. This obviously seems to be much more sensible. A mixture of the above mentioned operators is the *tournament selection*, where at first a small number of chromosomes

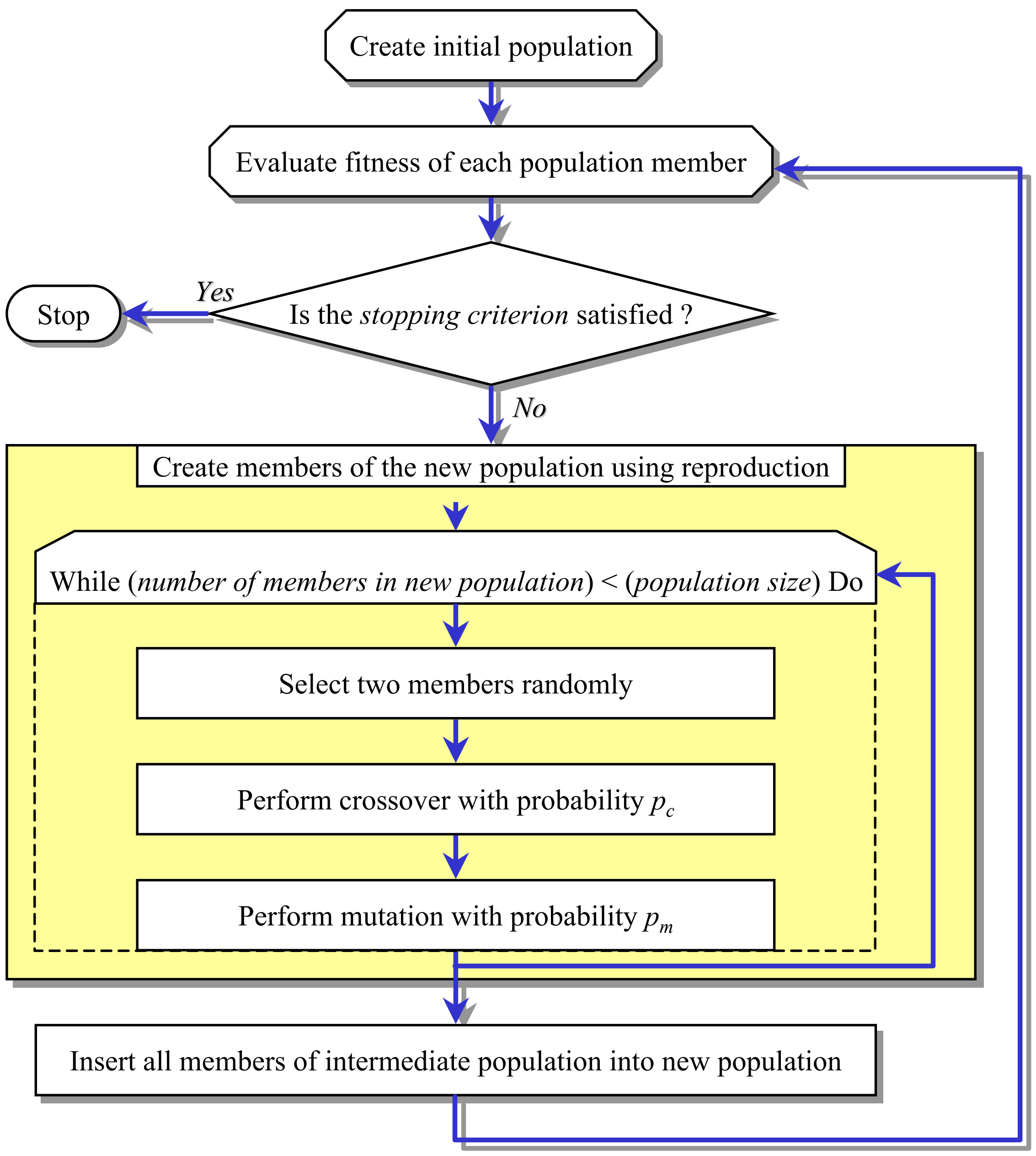

Create initial population
Evaluate fitness of each population member
Is the *stopping criterion* satisfied ?
Yes
Stop
No
Create members of the new population using reproduction
While (*number of members in new population*) < (*population size*) Do
Select two members randomly
Perform crossover with probability p_c
Perform mutation with probability p_m
Insert all members of intermediate population into new population

Figure 4: Simple flow chart of the standard genetic algorithm [2]. After the initialisation, the main loop evaluates the fitness of all population members, checks the stopping criterion, and creates a new population. The internal loop performs several genetic operations to create an intermediate population.

are uniformly selected and then compete with each other based on their fitness values. Finally, the winner of this tournament is selected. Standard *elitism* schemes should be applied as well, which pass the fittest individual of the current population to the next generation unmodified.

3.2 ANN-specific crossover

Once the intermediate population is complete, a new generation is created by applying several genetic operators to it. In standard implementations of evolutionary algorithms these operators are mainly *crossover* and *mutation*. Since it is the only possible way to exchange information between the individuals of a population and to pass information to the next generation, crossover is the most important step within the whole genetic algorithm. Thus, compared to all other described operators, crossover is probably the most frequently investigated one (see [25] for one possible example in this context). In the current implementation a standard algorithm has been selected as starting point: At first two chromosomes are randomly selected from the previously created intermediate population. Based on a predefined parameter, namely the crossover probability $p(c)$, it is decided whether these two chromosomes serve as parents and crossover takes place at all. If so, m-point uniform or linear interpolation crossover is performed, where typical values are $m \in \{1, 2, 3\}$.

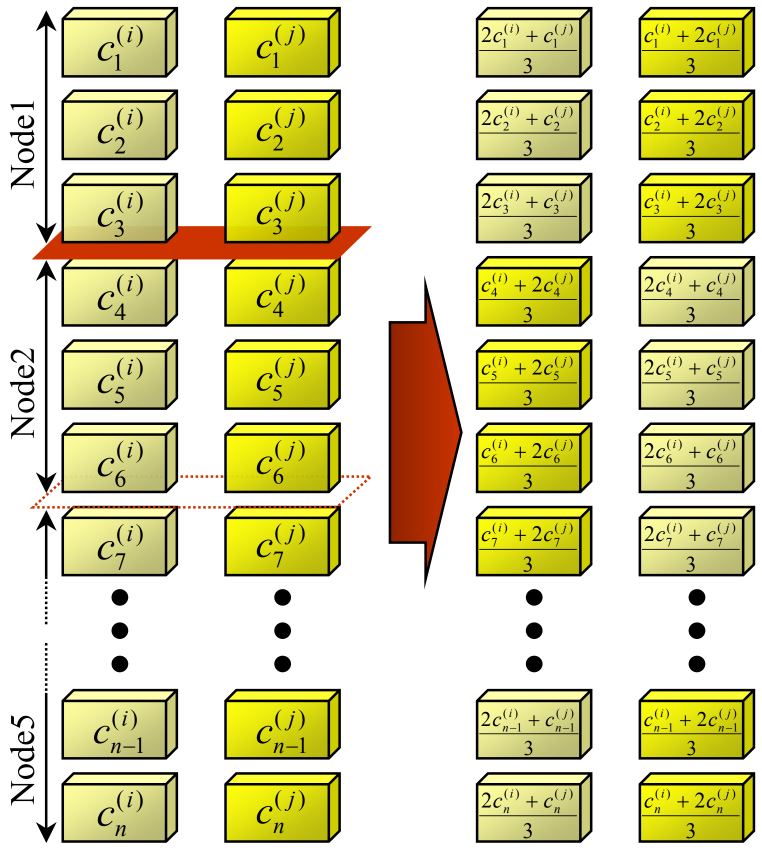

Node1
Node2
Node5
$c_1^{(i)}$
$c_1^{(j)}$
$\frac{2c_1^{(i)}+c_1^{(j)}}{3}$
$\frac{c_1^{(i)}+2c_1^{(j)}}{3}$
$c_n^{(i)}$
$c_n^{(j)}$

Figure 5: Example of *semantical* or ANN-specific crossover along with linear interpolation between parental genes. All potential crossover points are located according to the underlying *semantic* structure of the neural network *topology*.

Commonly, crossover points may potentially be located between any two neighbouring genes. In most cases, the actual crossover points are chosen randomly from them. If the genes carry rather unstructured information and their order within the chromosome does not really matter, this procedure seems to be applicable. However, in the present case each single gene represents one weight value of the neural network. The suggested mapping, as shown in Fig. 3, ensures that the weights belonging to the same neuron are located next to each other. Due to the dependence of a weight on its neighbours within the same neuron, it is rather unlikely that the performance of this neuron gets better and the overall network error is decreased, when only a few weights of this particular neuron are changed. Keeping this in mind, it seems to be advantageous to restrict potential crossover points to those locations separating entire neurons (Fig. 5). This might be called semantical or ANN-specific crossover.

In contrast to the standard operator this kind of a network specific crossover keeps the unity of weights belonging to the same neuron (see also [25]). Here, the approach of combining phenotypic and genotypic learning offers another advantage. It considers the phenotypic network topology, and thus the distribution or arrangement of the weights upon the networks's layout, on the one hand and the genotypic coding of the weights in the chromosome, as topological basis of all genetic operations, on the other, as a self-contained unit. In other words, the genetic algorithm is aware of the phenotypic relation of all genes within the chromosomes. As the result section will show, this generally leads to an improved performance of the learning process.

After crossover the *mutation* operator, depending on some predefined parameters (e.g., mutation probability $p(m)$), is applied. It is the only operator which can introduce *new* genetic information rather than just re-arranging it. Commonly a Gaussian distributed offset is added to each gene to be mutated. In [26] a mutation-based genetic neural network (MGNN) is proposed to replace Backpropagation by using the mutation strategy of local adaptation of evolutionary programming to effect the weight learning.

4 Results

4.1 A simple exclusive-or problem to demonstrate the basic properties

In order to have the results comparable to other studies and approaches, especially to conventionally trained MLPs, some frequently applied benchmarks have been selected to evaluate performance and properties of the evolutionarily trained ANN. One of the most frequently used benchmark applications to test MLPs is the *exclusive-or* (`xor`) problem and its extension to the continuous case. Furthermore, the system has been applied to substitute some conventionally trained MLP networks solving real-world problems in the field of classification tasks in image segmentation.

Generally it can be confirmed, that, as it would be expected, not all parameters of the GA have the same significant effect. The influence of the most important parameters, as seen from the angle of the user, is briefly discussed now. In the course of an extensive parameter variation some hundred networks were trained. Fig. 6 shows the typical training progress of six neural networks trained with different GA parameter sets. These results are alike to other studies dealing with similar tasks. Due to the high complexity of the system, some *similar* results could be achieved using *different* parameter sets. Changing two or more parameters might lead to an increase or decrease of a particular effect. Thus, the impact of a particular parameter could not be proved

Table 1: Standard parameter set used for reference training (see also Fig. 6 for a graph of the training progress) of `xor` with 2 inputs, 2 hidden neurons, and 1 output.

Parameter	Value
Neural activation function	$\tanh$
Population size	$pop\_size = 50$
Weight initialisation routine	Gaussian; $mean = 0.0$; $std = 5.0$
Stopping criterion	$network\_error = 0.01$; $max\_iter = 500$;
	$conv = 20$; $eps = 10^{-4}$
Fitness normalisation	norm; rank
Selection operator	roulette with elitism
Crossover	$p(c) = 0.8$; two-point; uniform;
	ANN-specific crossover points
Mutation	$p(m) = 0.1$; $mean = 0.0$; $std = 1.0$

in every combination. Based on a multitude of experiments a standard parameter set has been defined as reference (Tab. 1; Fig. 6, top left panel).

Both the average and minimal error is decreased in the progress of training. As to be expected, the average error starts at about 0.5 and is always higher than the minimal error. The fittest individual of the initial population has an error of 0.4. After 42 generations the error of the fittest chromosome falls below the desired threshold and the training is stopped at an error of 0.0061.

The population size is one of the primary parameters. Depending on the size (number of weights) of the neural network and the problem (complexity, dimensionality) to be solved by it, a range from $20 \dots 100$ individuals is sufficient. A smaller population size quickly leads to a steady state with identical chromosomes of an insufficient fitness (Fig. 6, top center). The training is cancelled at 500 iterations at an error of 0.25. On the other hand, a very large population entails not necessarily better results, but computation time and required resources increase. The danger of running into a steady state can be diffused by a relatively high mutation probability. That way more new genetic information is introduced to overcome this.

Besides general interaction of crossover rate and type concerning all genetic algorithms, the question where to set potential crossover points is in the context of neural network training of tremendous interest (see Sect. 3). For this reason ANN-specific crossover as suggested in Sect. 3.2 has been applied. Depending on the number of actual crossover points the probability to disrupt the weights belonging to the same neuron decreases with an increasing total amount of neurons within the network. In other words, especially in small neural networks the ANN-specific crossover is advantageous. For all evaluated parameter combinations the ANN-specific crossover resulted in a lead of $12\% \dots 47\%$. Thus it can beneficially contribute to the performance in addition to well chosen control parameters for selection and mutation.

The influence of several selection operators is shown in Fig. 6 (bottom row). Compared to the standard selection operator *roulette wheel*, the gap between minimal and average error is smaller, if *tournament selection* is used. The error threshold is met

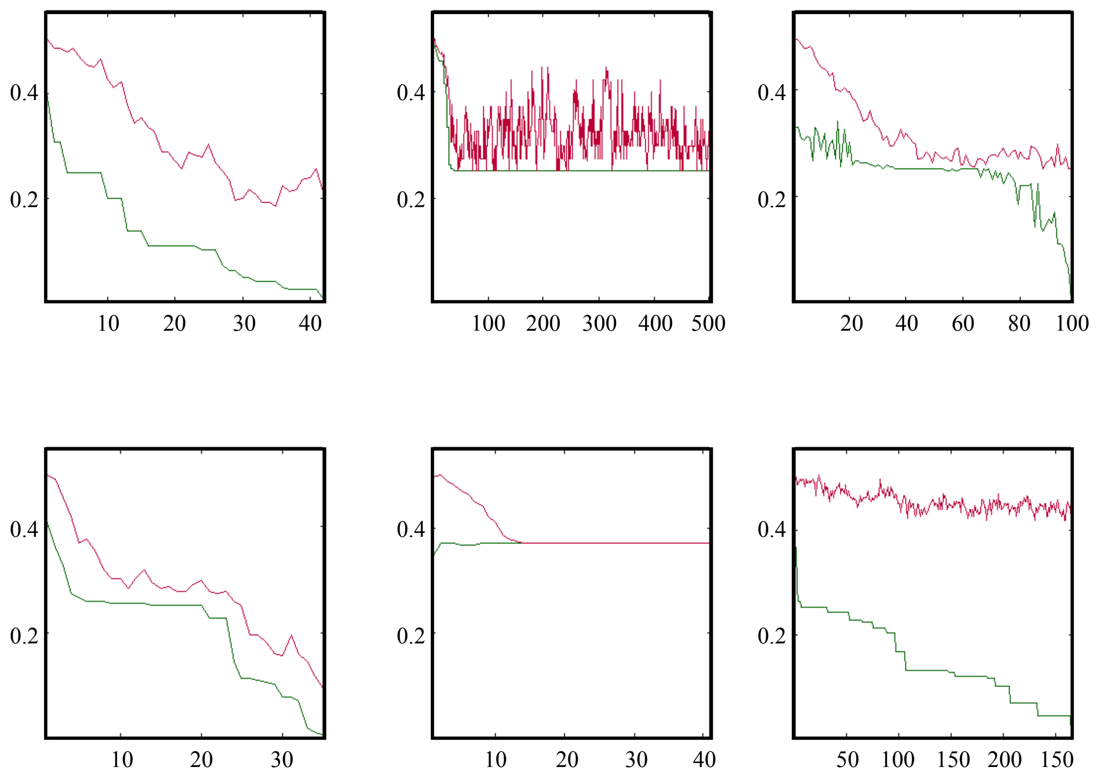


Figure 6: Training progress by means of recall error, scaled to $[0, 1]$, with the best weight set (lower line) and the average recall error (upper line). Several parameter combinations: Reference graph with standard parameters according to Tab. 1 (top left); population size reduced to 10 (top center); mutation disabled (top right); tournament selection (bottom left); uniform selection (bottom center); elitism disabled (bottom right). Note the variable scaling of the x-axes, showing the number of generations.

slightly earlier. Since *uniform selection* is not based on the fitness values and thus ranking is not applicable, the average error decreases hardly and the minimal error decreases very slowly. The probability, that the fittest chromosome, saved by elitism, finds another fit partner for reproduction is less.

If *mutation* is disabled (top right) minimal and average error meet each other quickly. That means, all chromosomes are identical. Without mutation there was no way out. In general, mutation should be considered in its close interaction between mutation probability $p(m)$ and mutation type (e.g., Gaussian). If the mutation values have a high impact on the chromosomes, i.e. when adding Gaussian distributed values with a high standard deviation, $p(m)$ should be reduced.

4.2 Generalizing the results

Similar behaviour could be observed for both the `continuous xor` and a number of real-world image classification (feature based image segmentation) tasks.

There is of course a rather strong correlation between the complexity of the problem to be solved by the ANN, which directly manifests oneself in the length of the chromo-

Table 2: Impact of maintaining phenotypic-genotypic-learning coherence (ANN-specific crossover) on the performance of the network training. Using a standard parameter set, derived from Tab. 1 and modified according to the network size, neural networks of different size and complexity have been trained either with ANN-specific crossover or without. The shown results are based on the number of generations that were necessary to obtain a particular network error and on statistical 10-fold cross validation.

Network size (input – hidden – output)	Generations **without** ANN-specific crossover	Generations **with** ANN-specific crossover
$2-2-1$	43 ± 8	30 ± 5 improved: 30%
$4-3-3$	62 ± 10	44 ± 9 improved: 29%
$18-12-8-4$	121 ± 26	96 ± 16 improved: 21%
$68-40-20$	148 ± 32	116 ± 19 improved: 22%
$68-50-30-20$	173 ± 41	142 ± 20 improved: 18%
$192-128-64-8$	315 ± 102	291 ± 82 improved: 7.6%
$192-16-8$	764 ± 524	371 ± 117 improved: 51%

some, and the required training time, in terms of both the *number* of generations and the computation time to run each *single* generation. The particular impact of a number of control parameters on the training performance, as shown in the previous subsection, is more or less evident as well. Much more interesting is the influence of the phenotypic-genotypic-learning coherence (ANN-specific crossover). This is shown in Tab. 2.

The results in Tab. 2 clearly confirm the presumption that maintaining an ANN-specific crossover is generally beneficial and that its impact is more visible in smaller networks. Compared to randomly selected crossover points the number of generations necessary to obtain a particular error value is always smaller and seems to asymptotically reach it as the network size increases.

The last network impressively shows the benefit for networks with very long weight vectors. Here, there are just 16 neurons of the hidden layer each of them having a weight vector of length 192. Obviously, this network needs much more learning cycles than a larger one to obtain the same error threshold, but may serve as a very good example to demonstrate the properties of ANN-specific crossover. Based on the mean number of cycles there is a lead of about 50% compared to the randomly selected crossover points.

5 Conclusion

This paper demonstrates the advantages of combining phenotypic and genotypic learning by means of substituting Backpropagation by a genetic algorithm *that is aware of and observes the coded topology of the neural network when applying its crossover operator*. Crossover points are chosen along the chromosome *according to the underlying structure of neural weights* rather than randomly. This prevents the phenotypic weight vector of any neuron to get disrupted by the recombination of the genotype.

This approach appeared to be most beneficial in smaller networks or in networks with rather long weight vectors, where it is more likely that the weights belonging to the same neuron could get disrupted by the genetic recombination operator. Since no negative effects appeared and the additional cost of this approach is pretty small (the topology information of the network is available anyway), it seems to be worth to be considered as default way to design the recombination operator of genetic algorithms in case of their application to neural network training.

Future work will focus on the extension of this approach to further supervised trained neural networks that are likely to benefit from this as well.

References

[1] J. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[2] A. van Rooij, L. Jain, and R. Johnson, *Neural Network Training Using Genetic Algorithms*. Singapore: World Scientific, 1996.

[3] U. Seiffert, “Training of large-scale feed-forward neural networks,” in *Proc. Int. Joint Conf. on Neural Networks*. IEEE Press, 2006, pp. 10 780–10 785.

[4] T. Czauderna and U. Seiffert, “Implementation of MLP networks running Backpropagation on various parallel computer hardware using MPI,” in *Proc. Int. Conf. on Recent Advances in Soft Computing*, 2004, pp. 116–121.

[5] R. Lippmann, “An introduction to computing with neural nets,” *IEEE ASSP Magazine*, vol. 4, pp. 4–23, 1987.

[6] G. Miller, P. Todd, and S. Hegde, “Designing neural networks using genetic algorithms,” in *Proc. Int. Conf. on Genetic Algorithms*, 1989, pp. 379–385.

[7] K. Balakrishnan and V. Honavar, “Properties of genetic representations of neural architectures,” in *Proc. World Conf. on Neural Networks*, Washington, DC., USA, 1995, pp. 807–813.

[8] B. Sendhoff and M. Kreutz, “Evolutionary optimization of the structure of neural networks by a recursive mapping as encoding,” in *Proc. Int. Conf. on Artificial Neural Nets and Genetic Algorithms*, Norwich, U.K., 1998, pp. 368–372.

[9] V. Bevilacqua, G. Mastronardi, F. Menolascina, P. Pannarale, and A. Pedone, "A novel multi-objective genetic algorithm approach to artificial neural network topology optimisation: The breast cancer classification problem," in *Proc. Int. Joint Conf. on Neural Networks*. IEEE Press, 2006, pp. 1958–1965.

[10] P. Robbins, A. Soper, and K. Rennolls, "Use of genetic algorithms for optimal topology determination in back propagation neural networks," in *Proc. Int. Conf. on Artificial Neural Nets and Genetic Algorithms*, Innsbruck, Austria, 1993, pp. 726–730.

[11] P. Angeline, G. Saunders, and J. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Trans. Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994.

[12] A. Hämäläinen, "Using genetic algorithms in Self-Organizing Map design," in *Proc. Int. Conf. on Artificial Neural Networks and Genetic Algorithms*, Ales, France, 1995, pp. 364–367.

[13] H. Kitano, "Neurogenetic learning: An integrated method of designing and training neural networks using genetic algorithms," *Physica D*, vol. 75, pp. 225–228, 1994.

[14] J. Branke, "Evolutionary algorithms for neural network design and training," University of Karlsruhe, Institute AIFB, Tech. Rep. 322, 1995.

[15] F. Leung, H. Lam, S. Ling, and P. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Trans. Neural Networks*, vol. 14, no. 1, pp. 79–88, 2003.

[16] J. Tsai, J. Chou, and T. Liu, "Tuning the structure and parameters of a neural network by using hybrid Taguchi-genetic algorithm," *IEEE Trans. Neural Networks*, vol. 17, no. 1, pp. 69–80, 2006.

[17] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[18] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psych. Rev.*, vol. 65, pp. 386–408, 1958.

[19] D. Parker, "Learning-logic," MIT, Center for Computational Research in Economics and Management Science, Tech. Rep. TR47, 1985.

[20] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. R. et al., Ed. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[21] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, pp. 145–151, 1999.

[22] Y. Shang and B. Wah, "Global optimization for neural network training," *Computer*, vol. 29, no. 3, pp. 45–54, 2003.

[23] A. Torn and A. Zilinskas, *Global Optimization*. New York, NY, USA: Springer-Verlag New York, Inc., 1989.

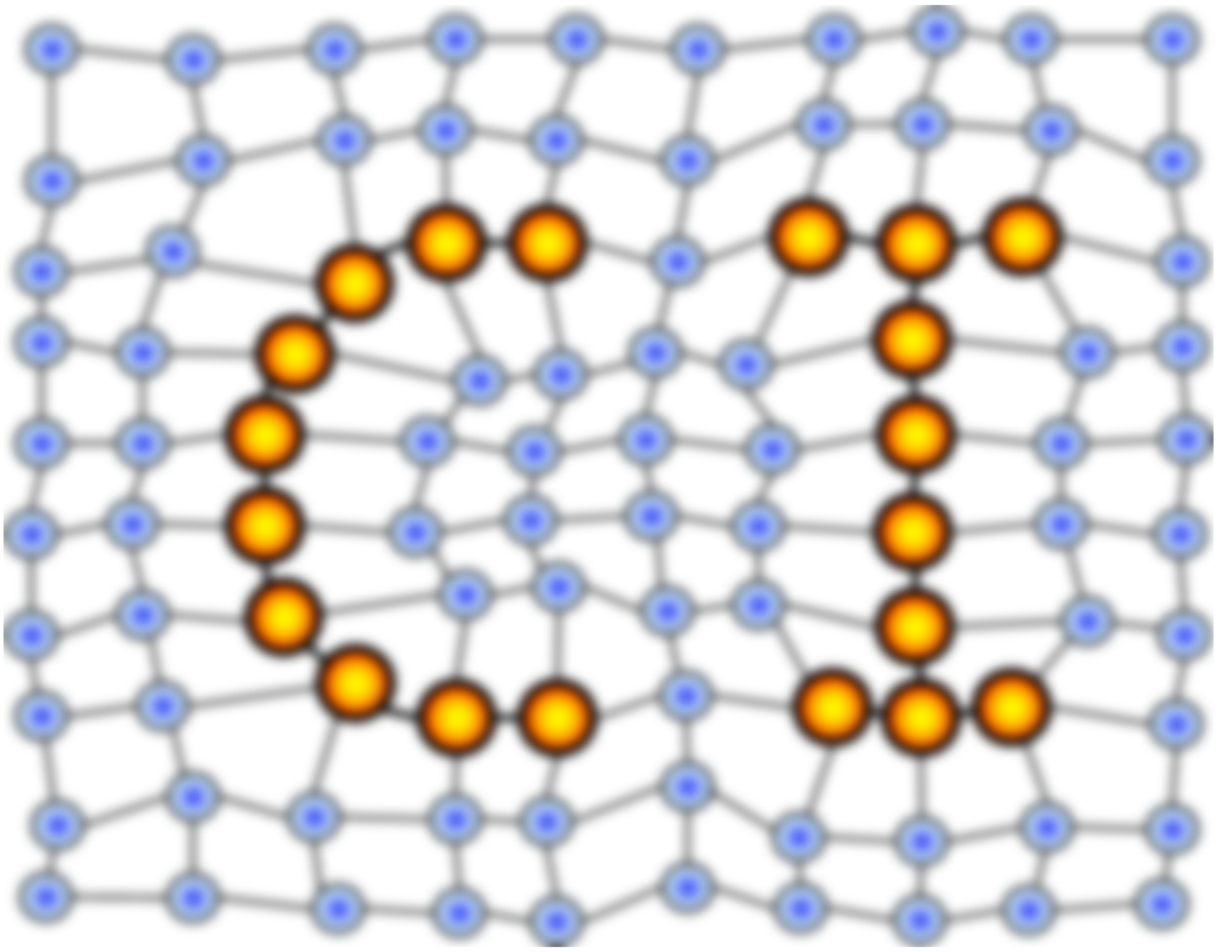
[24] U. Seiffert, “Multiple Layer Perceptron training using Genetic Algorithms,” in *Proc. European Sympos. on Artificial Neural Networks*, 2001, pp. 159–164.

[25] N. García-Pedrajas, D. Ortiz-Boyera, and C. Hervás-Martínez, “An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization,” *Neural Networks*, vol. 19, no. 4, pp. 514–528, 2006.

[26] P. Palmes, T. Hayasaka, and S. Usui, “Mutation-based genetic neural network,” *IEEE Trans. Neural Networks*, vol. 16, no. 3, pp. 587–600, 2005.

MACHINE LEARNING REPORTS

Report 01/2008



Impressum

Machine Learning Reports ISSN: 1865-3960

▽ Publisher/Editors

PD. Dr. rer. nat. Thomas Villmann & Dr. rer. nat. Frank-Michael Schleif
Medical Department, University of Leipzig
Semmelweisstrasse 10, D-04103 Leipzig, Germany •
http://www.uni-leipzig.de/˜compint

▽ Acknowledgments

We would like to thank the reviewers for their time and patience.