Using Genetic Algorithms for Automatic Construction of Higher-Order Neural Models

Byoung-Tak Zhang Heinz Mühlenbein
German National Research Center for Computer Science (GMD)
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
E-mail: zhang@gmd.de, muehlen@gmd.de

Abstract— **Genetic algorithms have been used as a method for search, optimization and machine learning. As opposed to conventional learning algorithms for neural networks, genetic algorithms make relatively few assumptions about the network types. Thus they are useful for the development of novel neural network architectures. In this paper we describe a genetic method for breeding higher-order neural archiectures based on noisy data. The crux of the method is the weighted tree representation, what we call neural trees, and the Occam's razor for the construction of minimal neural models. The effectiveness of the method is shown by experimental results and the analysis of the fitness landscape.**

1 Introduction

One of the most popular architectures used for supervised learning applications has been the mutilayer feedforward network. A commonly adopted topology employs one hidden layer with full connectivity between neighboring layers. This structure has been very successful for many applications. However, they have some weaknesses. For instance, the fully connected structure is not necessarily a good topology unless the task contains a good predictor for the full input space.

Most network architectures consist of neural units which compute the weighted sum of inputs. These summation units are especially appropriate to approximating additive functions, since they employ linear combinations of inputs. However, the multilayer perceptrons cannot approximate efficiently if there are high order interactions between the inputs. Adding additional hidden layers may help to extend the representational capacity of the network, but the training becomes more difficult. A solution may be to use higher-order units. The necessity and usefulness of higher-order neural networks have been well-known. However the explosive number of terms hampers the design and training of such networks.

In this paper we present a method for the construction of higher-order neural networks with partial connectivity. The method uses a genetic algorithm. Genetic algorithms are search methods based on a population of individuals, each of which represents a search point in the space of potential solutions to a given problem [2, 5, 14, 18]. The population is arbitrarily initialized, and it evolves toward better and better regions of the search space by means of randomized processes of selection, mutation and recombination. The evironment delivers the fitness value of the search points, and the selection process favors those individuals of higher fitness to reproduce more often than those of lower fitness. The recombination mechanism allows the mixing of parental information while passing it to their descendants, and mutation introduces innovation into the population.

The method uses a tree representation of the network, called neural trees, on which genetic operators are applied to modify and find fitter architectures. Another feature of the method is the use of Occam's razor in its fitness function. It makes an optimal trade-off between the error fitting ability and the parsimony of the network. In section 2 we describe this representation scheme in more detail. Section 3 describes the evolutionary learning algorithm and shows experimental results on the synthesis of higher-order neural networks from noisy data. Section 4 studies the effectiveness of Occam's razor by analyzing the fitness landscape of the problem. Future extensions of current work are described in section 5.

2 Neural Trees

In weight optimization, the set of weights is represented as a chromosome and a genetic search is applied on the encoded representation to find a set of weights that best fits the training data. Some encouraging results have been reported which are comparable with conventional learning algorithms [9]. Where gradient or error information is not available, genetic algorithms may be a promising training method. In architecture optimization, the topology of the networks is encoded as a chromosome and some genetic operators are applied to find an architecture which best fits the specified task according to some explicit design criteria. Many methods have been proposed for evolving network topologies.

A general way of evolving genetic neural networks was suggested by Mühlenbein and Kindermann in [13]. Recent works, however, have focused on using genetic algorithms separately in each optimization problem, mainly in optimizing the network topology. Harp *et al.* [4] and Miller *et al.* [8] have described representation schemes in which the anatomical properties of the network structure are encoded as bit-strings. Similar representation has also been used by Whitley *et al.* [19] to prune unnecessary connections.

Kitano [6] suggested encoding schemes in which a network configuration is indirectly specified by a graph generation grammar which is evolved by genetic algorithms. All these methods use the backpropagation algorithm [16], a gradient-descent method, to train the weights of the network.

We represent a feedforward network as a set of m trees, each corresponding to one output unit. For example, the genotype of a feedforward network consisting of $n = 6$ inputs and $m = 1$ output unit is encoded as:

$$
\begin{array}{l}
(\ U_1\ \theta_1\ w_{11}\ (\ U_2\ \theta_2\ w_{21}\ (\ x_2\ )\ w_{22}\ (\ x_3\ )\ ) \\
\quad w_{12}\ (\ U_3\ \theta_3\ \ w_{31}\ (\ x_4\ ) \\
\qquad\qquad w_{32}\ (\ U_4\ \theta_4\ w_{41}\ (\ x_1\ )\ w_{42}\ (\ x_3\ )\ w_{43}\ (\ x_6\ )\ )\ ) \\
\quad w_{13}\ (\ x_1\ ) \\
\quad w_{14}\ (\ U_5\ \theta_5\ w_{51}\ (\ x_2\ )\ w_{52}\ (\ x_4\ )\ w_{53}\ (\ x_5\ )\ )\ )
\end{array}
$$

Figure 1 shows the corresponding tree representation. In this tree representation, what we call neural tree, a node consists of one or more elements. For hidden and output units, each node contains an activation function type U_i, a threshold value θ_i, and an arbitrary number of weight values w_{ij}. The node may point recursively to other hidden units U_i or an external input unit x_k, $k \in \{1, \cdots, n\}$.

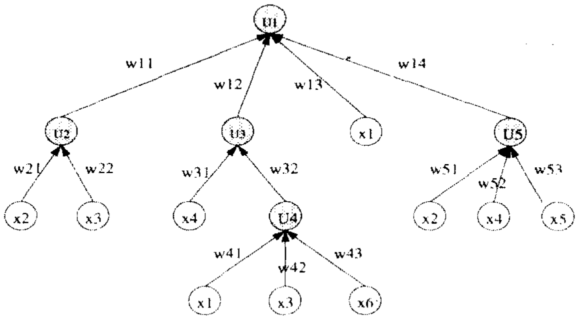

Figure 1: *Tree representation of the network. For simplicity, the thresholds are not shown.*

This encoding scheme can represent any feedforward network with local receptive fields and direct connections between non-neighboring layers (see [21] for more details) and thus extends the tree representation used in [7]. This is contrasted with the more commonly used perceptron architecture of fully connected feedforward networks.

3 Evolving Sigma-Pi Neural Trees

Note also that this representation allows any type of activation functions to be defined. In particular, we may use pi units as well as the usual sigma units. While a sigma-unit calculates a *sum* of weighted inputs,

$$u_i = \sum_{j \in R(i)} w_{ij} x_j, \tag{1}$$

a pi-unit builds a *product* of weighted inputs:

$$u_i = \prod_{j \in R(i)} w_{ij} x_j. \tag{2}$$

Here w_{ij} is the connection weight from unit j to unit i and $R(i)$ denotes the receptive field of unit i. By using pi units we can, for example, directly build kth-order terms

$$T^{(k)} = f_k \left(\prod_{j=1}^{j_k} w_{kj} x_j \right), \tag{3}$$

which in conventional neural networks require a number of layers consisting of summation units. The higher-order terms can be again used as building blocks which are able to capture a high-order correlational structure of the data. In particular, by building a sigma unit which has as input various higher-order terms $T^{(k)}$, a higher-order neural tree of sigma-pi units can be constructed:

$$y_i = f_i(u_i) = f_i \left(\sum_k w_{ik} T^{(k)} \right) = f_i \left(\sum_k w_{ik} f_k \left(\prod_{j=1}^{j_k} w_{kj} x_j \right) \right). \tag{4}$$

For the construction of neural models we maintain a population $\mathcal{A}$ consisting of M individuals A_i of

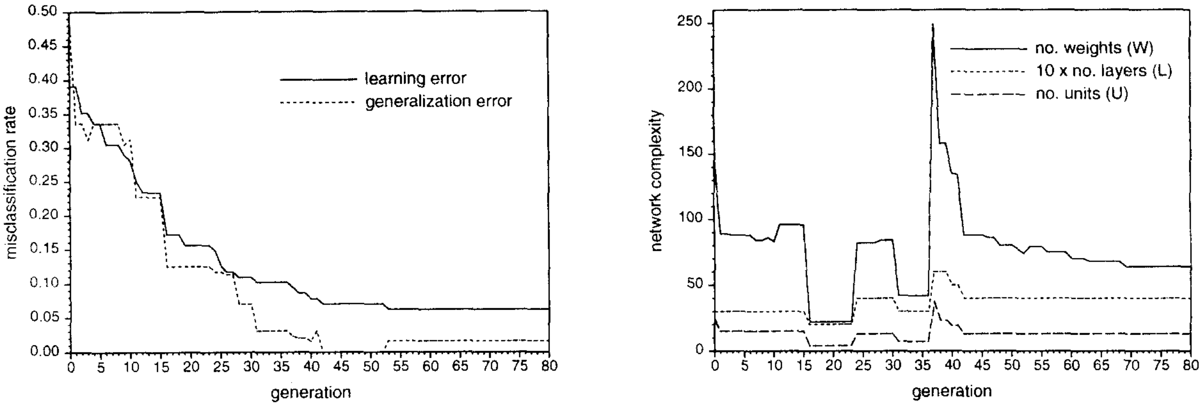

Figure 2: *Evolving sigma-pi networks from noisy data*

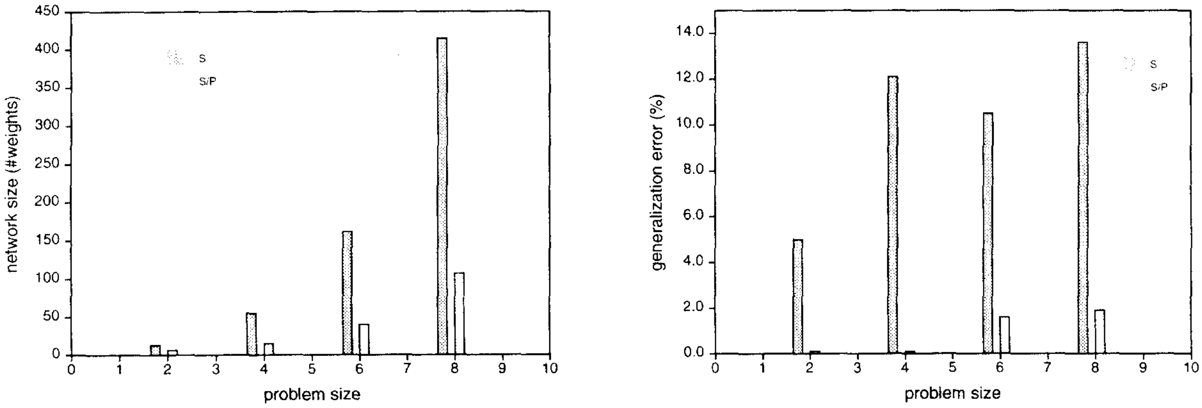

Figure 3: *Performance comparison of perceptrons and sigma-pi networks*

variable size. Each individual is a neural network represented as a set of neural trees. The algorithm is summarized as follows:

1. Generate initial population $\mathcal{A}(0)$ of M networks at random. Set current generation $g \leftarrow 0$.
2. Evaluate fitness values $F_i(g)$ of networks using the training set of N examples.
3. If the termination condition is not satisfied, continue with step 4.
4. Select upper τM networks of gth population into the mating pool $\mathcal{B}(g)$.
5. Each network in $\mathcal{B}(g)$ undergoes a local hillclimbing, resulting in revised mating pool $\mathcal{B}(g)$.
6. Create $(g+1)$th population $\mathcal{A}(t+1)$ of size M by applying genetic operators to randomly chosen parent networks in $\mathcal{B}(g)$.
7. Replace the worst fit network in $\mathcal{A}(t+1)$ by the best in $\mathcal{A}(t)$.
8. Set $g \leftarrow g+1$ and return to step 2.

We use a training set D to measure the fitness F_i of the network A_i with weights $\mathbf{w}$:

$$F_i = F(D|\mathbf{w}, A_i) = \frac{E(D|\mathbf{w}, A_i)}{m \cdot N} + \frac{C(\mathbf{w}|A_i)}{N \cdot C_{max}}, \tag{5}$$

where N is the number of training examples. A theoretical background behind this fitness function is discussed in [21]. The first term expresses the accuracy penalty caused by the error for the training set. The second term in the fitness function expresses the complexity penalty of the network, reflecting the principle of Occam's razor.

The method was used to synthesize sigma-pi networks. The task was the parity problem of input size 8. A total of 128 correct examples were generated randomly to get a training set and then noise was inserted to this data by randomly changing the ouput value with 5% probablity. The generalization performance of the best solution in each generation was tested by the complete data set of $2^8 = 256$ noise-free examples. The population was initialized by allowing every individual to contain sigma and pi units with 50% probability each. The depth of the initial networks was limited to 3. The truncation rate τ was 40%.

Figure 2 (left) shows a typical evolution of the training and generalization error of the best fit sigma-pi networks. In spite of the noise, a good correspondence is observed between learning and generalization performance. Figure 2 (right) shows the corresponding evolution of the complexity of the best fit network in each generation. Notice that the change of network performance is closely related to the change of its complexity.

We also studied the scalability of the method to synthesize the sigma-pi network architecture. For these experiments we used noise-free data consisting of 2^n examples for parity problems of input size $n = 2, 4, 6, 8$. For each problem size n, we used the population size $M = 100n/2$ and the maximum generation $g_{max} = 10n$. For example, $M = 400$ and $g_{max} = 80$ for $n = 8$. To test the effectiveness of pi units for solving the parity problem we also ran experiments using only sigma units. The experiments were the same as before except that all units are initialized as sigma units. The average performances are compared in Figure 3. As the results show, the additional use of pi units consistently improved the performance in accuracy as well as in complexity reduction.

4 Fitness Landscape and Occam's Razor

We analyzed the relationship between the average generalization performance and complexity of sigma-pi neural trees. The parity problem of input size $n = 7$ was used in this analysis. We first generated a clean data set B_N of size $N = 2^7$. A noisy training set D_N was then generated from B_N by flipping the output value of each example with 5% probability.

Approximately $M = 20000$ sigma-pi networks of differing size and weights were generated. The performance of each network A_i with weights $\mathbf{w}$ was then measured on the noisy data sets D_N:

$$E(D_N|\mathbf{w}, A_i) = \sum_{c=1}^{N}\sum_{k=1}^{m}(y_{ck} - f_k(\mathbf{x}_c; \mathbf{w}, A_i))^2. \tag{6}$$

where y_{ck} and $f_k(\mathbf{x}_c; \mathbf{w}, A_i)$ are the desired and actual value of the kth output of the ith network given the input pattern $\mathbf{x}_c$. Likewise, the generalization performance $E(B_N|\mathbf{w}, A_i)$ of the network was measured on the test set B_N of N clean examples. Each performance measure was then normalized to be in the interval $[0, 1]$ by

$$L(i) = \frac{1}{mN}E(D_N|\mathbf{w}, A_i), \tag{7}$$

$$G(i) = \frac{1}{mN}E(B_N|\mathbf{w}, A_i). \tag{8}$$

For each network we also measured its complexity in terms of the number of binary weights:

$$W_i = W(\mathbf{w}|A_i) = \sum_{j,k} w_{jk}^2, \tag{9}$$

where the indices j and k run over all the units in A_i. We then depicted the performance as a function of the network complexity by computing the average training errors $E_L(\omega)$, average generalization errors $E_G(\omega)$ and their difference $E_{G-L}(\omega)$ for each ω:

$$E_L(\omega) = \text{avg}\{L(i) \mid W_i = \omega,\ i = 1, \cdots, M\}, \tag{10}$$

$$E_G(\omega) = \text{avg}\{G(i) \mid W_i = \omega,\ i = 1, \cdots, M\}, \tag{11}$$

$$E_{G-L}(\omega) = E_G(\omega) - E_L(\omega). \tag{12}$$

The resulting fitness landscape is depicted in Figure 4 (left). The graphs are drawn for the ω-points at which more than five W_i-instances have been found in computing (10) and (11).

Similarly we analyzed the fitness landscape as a function of the number of units in the network, as shown in Figure 4 (right). The figures show the general tendency that the relative generalization error increases as the network size grows; that is, generalization ability of the network is inversely proportional to network complexity.

The effect of Occam's razor was investigated on the parity problem of size $n = 6, 7, 9$. For each problem size n we generated a training set of $2^n/2$ examples which were chosen randomly and inserted noise by changing the ouput value with 5% probability. The generalization performance of the best solution in each generation was tested by the complete data set of 2^n noiseless examples. The population was initialized for every individual to contain sigma and pi units with 50% probability each. The depth of initialized network was limited to 3. The truncation rate was 40%. For each problem size n, we used the population size $M = 100n/2$ and the maximum generation $g_{max} = 10n$. For example, $M = 300$ and $g_{max} = 60$ for $n = 6$.

The performance of the method with Occam's razor, i.e. fitness function (5), was compared with a method that uses just the error term as the fitness measure, i.e. $F(D|\mathbf{w}, A) = E(D|\mathbf{w}, A)/mN$. Both methods used the same data. For each method, several runs were executed to observe the complexity of the best solution and its training and generalization performance (Figure 5).

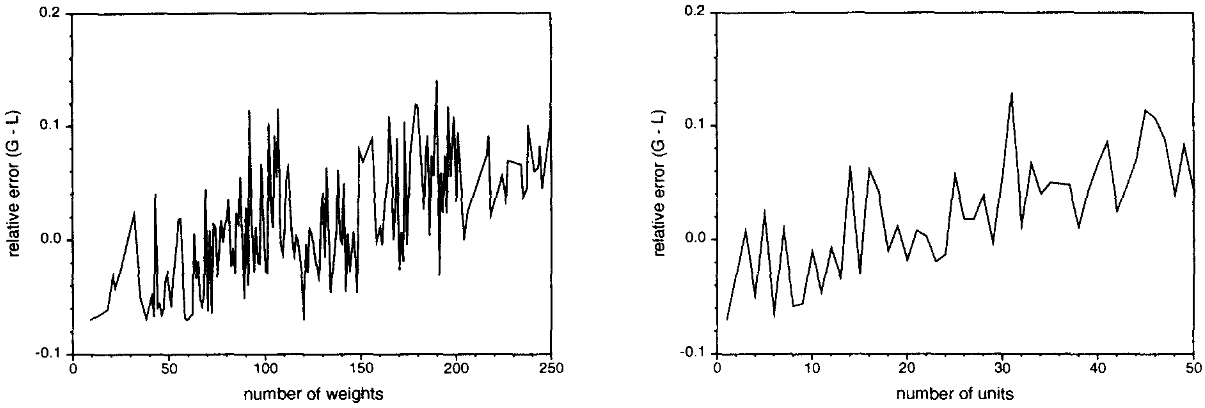

Figure 4: *Relationship between network size and generalization performance*

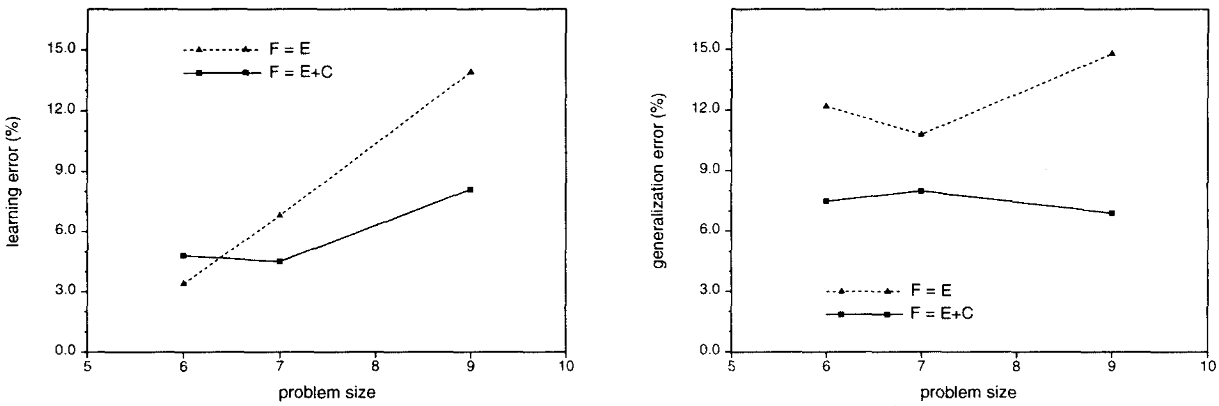

Figure 5: *Effects of Occam's razor on learning and generalization accuracy*

The results show that applying Occam's razor achieves significantly better generalization performance as was expected by the landscape analysis. It is also interesting to note that solutions with Occam's razor achieved better learning performance than without it for $n > 6$. This is because the search with Occam's razor focuses on a smaller search space while the search without it may explore too large a space to be practical. Since the evolution time is limited to the maximum of g_{max} generations, using Occam's razor can find a solution faster than without it.

5 Summary and Conclusion

Both genetic algorithms and neural networks are computational paradigms that, while not actually biological, are very loosely based on biological concepts. Idealized mathematical models are used to understand the computational behavior of simple systems that in a very general way imitate computations found in nature. The fields of genetic algorithms and neural networks are similar inasmuch as both have roots in the 1950s and 1960s and both have in some sense been rediscovered, and in many ways redefined, in the last decade. Combining genetic algorithms and neural networks has a biological motivation: After all, the genetic code that is fundamental to all higher animal species was involved in the evolution of biological neural systems. But, just as we are still far from fully understanding biological neural systems, we are also yet far from fully understanding biological genetic codes. From the engineering point of view, various schemes for combining genetic algorithms and neural networks have been proposed and tested in recent years [17], including preprocessing of data for neural network application, determination of neural network weights, and optimization of the network topology.

In this paper we have presented an evolutionary method for constructing higher-order neural network architectures. The method uses a tree encoding scheme in which the node type, weight, size and topology of the network are dynamically adapted by genetic operators. We demonstrate the effectiveness of the genetic algorithm on the synthesis of sigma-pi neural networks which are useful for building higher-order terms. In particular, we show how the parity problem can be solved more efficiently by sigma-pi networks than by multilayer perceptrons. We also studied the fitness landscape of sigma-pi neural networks to see the relationship between the solution complexity and its generalization ability. The comparison of learning and generalization performance as a function of network complexity suggests the usefulness of Occam's razor. In contrast to conventional learning algorithms for neural networks, the presented method makes relatively few assumptions on the architecture space in which the search is performed. Thus it may be used to breed other types of neural architectures. The potential for evolving novel neural architectures that are customized for specific applications is one of the most interesting properties of genetic algorithms.

Acknowledgement

This research was supported in part by the Real-World Computing Program under the project SIFOGA.

References

[1] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, vol. 1, pp. 1–23, 1993.

[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1989.

[3] F. Gruau, "Genetic synthesis of modular neural networks," in *Proc. Fifth Int. Conf. Genetic Algorithms*, Morgan Kaufmann, 1993, pp. 318–325.

[4] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," in *Proc. Third Int. Conf. Genetic Algorithms*, Morgan Kaufmann, 1989, pp. 360–369.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

[6] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," *Complex Systems*, vol. 4, pp. 461–476, 1990.

[7] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[8] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proc. Third Int. Conf. Genetic Algorithms*, Morgan Kaufmann, 1989, pp. 379–384.

[9] D. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *Proc. Int. Joint Conf. Artificial Intelligence*, 1989.

[10] H. Mühlenbein, "Evolution in time and space—The parallel genetic algorithm," in *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991, pp. 316–338.

[11] H. Mühlenbein, "Evolutionary algorithms: Theory and applications," in *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (eds.) Wiley, 1993.

[12] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm I: Continuous parameter optimization," *Evolutionary Computation*, vol. 1, pp. 25–49, 1993.

[13] H. Mühlenbein and J. Kindermann, "The dynamics of evolution and learning—Towards genetic neural networks," in *Connectionism in Perspective*, R. Pfeifer *et al.* (eds.) Elsevier, 1989, pp. 173–197.

[14] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog, 1973.

[15] J. Rissanen, "Stochastic complexity and modeling," *The Annuls of Statistics*, vol. 14, pp. 1080–1100, 1986.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error-propagation," in *Parallel Distributed Processing*, Vol. I, D. E. Rumelhart and J. L. McClelland (eds.) MIT Press, 1986, 318–362.

[17] J. D. Schaffer, D. Whitley, and L. J. Eshelman, "Combinations of genetic algorithms and neural networks: A survey of the state of the art," in *Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks*, IEEE, 1992, pp. 1–37.

[18] H.-P. Schwefel, *Numerical Optimization of Computer Models*. Chichester: Wiley, 1981.

[19] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: Optimizing connections and connectivity," *Parallel Computing*, vol. 14, pp. 347–361, 1990.

[20] B. T. Zhang and H. Mühlenbein, "Genetic programming of minimal neural nets using Occam's razor," in *Proc. Fifth Int. Conf. Genetic Algorithms*, S. Forrest (ed.) Morgan Kaufmann, 1993, pp. 342–349.

[21] B. T. Zhang and H. Mühlenbein, "Evolving optimal neural networks using genetic algorithms with Occam's razor," forthcoming in *Complex Systems*, 1994.

[22] B. T. Zhang and H. Mühlenbein, "Synthesis of sigma-pi neural networks by the breeder genetic programming," in *Proc. IEEE World Congress on Computational Intelligence*, IEEE, 1994.

[23] B. T. Zhang and G. Veenker, "Neural networks that teach themselves through genetic discovery of novel examples," in *Proc. Int. Joint Conf. Neural Networks*, IEEE, 1991, vol. I, pp. 690–695.