Synthesis of Sigma-Pi Neural Networks by the Breeder Genetic Programming

Byoung-Tak Zhang and Heinz Mühlenbein

Abstract— **Genetic programming has been successfully applied to evolve computer programs for solving a variety of interesting problems. In the previous work we introduced the breeder genetic programming (BGP) method that has Occam's razor in its fitness measure to evolve minimal size multilayer perceptrons. In this paper we apply the method to synthesis of sigma-pi neural networks. Unlike perceptron architectures, sigma-pi networks use product units as well as summation units to build higher-order terms. The effectiveness of the method is demonstrated on benchmark problems. Simulation results on noisy data suggest that BGP not only improves the generalization performance, it can also accelerate the convergence speed.**

I. Introduction

Genetic programming has been successfully used to evolve computer programs for solving many interesting problems in artificial intelligence and artificial life [4, 16, 18, 19]. Similar to usual genetic algorithms (GAs), genetic programming (GP) starts with a population of randomly generated individuals. Each individual is a program that, when executed, is the candidate solution to the problem. These programs are expressed as parse trees, or LISP S-expressions. Fitness proportionate selection and crossover are used to produce increasingly fitter populations of computer programs.

While most GAs use binary strings of fixed size [9], GPs use structured representations of variable length. This is important because it is particularly suited to problems in which the optimal underlying structure must be discovered. One problem with the variable length is that the program size may grow without bound. For example, Kinnear [16] reports that all but a very few of evolved solutions to his sorting problems were so large as to defy any human understanding of them. Tackett [29], in his application of genetic programming to image processing tasks, oberserves that the size and complexity of trees grows without performance improvement.

In the previous work [34, 35] we introduced the breeder genetic programming (BGP) that employs Occam's razor in its fitness measure to evolve optimal or minimal size multilayer perceptrons. We apply here the BGP method to synthesize sigma-pi neural networks. Unlike multilayer perceptrons, sigma-pi networks use product units as well as summation units to build higher-order terms. In section II we illustrate the usefulness of higher-order terms and show how the sigma-pi networks can be used to represent higher-order networks. The representation scheme and the algorithm are described in section III. Simulation results are reported in section IV, followed by concluding remarks in section V.

The authors are with the German National Research Center for Computer Science (GMD), Schloss Birlinghoven, D-53757 Sankt Augustin, Germany. E-mail: {zhang, muehlen}@gmd.de

II. Sigma-Pi Neural Networks

To motivate the approach, we start with a brief description of multilayer neural networks. Multilayer perceptrons are feedforward networks with one or more layers of nodes between the input and output units. These additional layers contain hidden units that are not directly connected to both the input and output units. The input-output relation of the units is given in these networks by weighted sum of inputs

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

where w_{ij} is the connection weight from unit j to unit i and $R(i)$ denotes the receptive field of unit i. The total input is then transferred to upper layer units by a nonlinear activation function f, e.g. a threshold function:

$$y_i = f(u_i) = \begin{cases} +1 & \text{if } u_i > \theta_i \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

where θ_i is a threshold.

A commonly adopted architecture consists of one hidden layer with full connectivity between neighboring layers. This structure has been very successful for many applications. However, they have some weaknesses:

1. The full connectivity between layers and often only between neighboring layers try to find a prediction over the full input space. This is not necessarily a good strategy. A particular task might not contain a good predictor for the full input space, but might contain functions capable of good prediction on specified regions of input space.
2. They 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.

In the previous work [34, 35]. we use genetic programming to construct a problem-specific architecture whose

0-7803-1899-4/94 $4.00 ©1994 IEEE

size and depth are adapted flexibly during evolution. The method allows partial connectivity and direct connections between non-neighboring layers to find a parsimonious architecture. In this architecture input units can be connected directly to output units. Although the method turned out to be useful to find parsimonious networks, the simulation results and the analysis of the landscapes suggested that the representation scheme does not scale well on parity-like problems.

To improve the scaling property, we extend in the present work the function set of the genetic programming to include pi-units as well as the sigma-units. While a sigma-unit calculates a sum of weighted inputs, a pi-unit builds a *product* of weighted inputs:

$$u_i = \prod_{j \in R(i)} v_{ij} x_j. \tag{3}$$

Here v_{ij} is the connection weight from unit j to unit i and $R(i)$ denotes the receptive field of unit i. The resulting total input is propagated to upper layer units by an activation function chosen depending on applications.

The pi-units has been suggested earlier in the neural network community [5, 6, 27] and employed in polynominal networks [7, 14] and higher-order networks [1, 8].

A higher-order neuron of order k has an input-output relation given by

$$\begin{aligned} y &= f(u), \\ u &= w_0 + \sum_i w_i^{(1)} x_i + \sum_{i_1} \sum_{i_2} w_{i_1 i_2}^{(2)} x_{i_1} x_{i_2} + \cdots \\ &\quad + \sum_{i_1} \sum_{i_2} \cdots \sum_{i_k} w_{i_1 \cdots i_k}^{(k)} x_{i_1} \cdots x_{i_k} \end{aligned} \tag{4}$$

where all indices $i_1, ..., i_m$ in $w_{i_1 \cdots i_m}^{(m)}$ are assumed to take different values satisfying $i_1 < i_2 < \cdots < i_m$.

Since the k-th order term consists of a linear weighted sum over k-th order products of inputs, we can rewrite it using pi-units:

$$\begin{aligned} T^{(k)} &= \sum_{i_1} \sum_{i_2} \cdots \sum_{i_k} w_{i_1 \cdots i_k}^{(k)} x_{i_1} \cdots x_{i_k} \\ &= \sum_{i_1} \sum_{i_2} \cdots \sum_{i_k} w_{i_1, \ldots, i_k}^{(k)} g\left(\prod_{i=i_1}^{i_k} x_i \right) \\ &= \sum_{i_1} \sum_{i_2} \cdots \sum_{i_k} w_{i_1, \ldots, i_k}^{(k)} g\left(\prod_{i=i_1}^{i_k} v_i x_i \right) \\ &= \sum_{(i_1, i_2, \ldots, i_k)} w_{i_1, \ldots, i_k}^{(k)} P^{(k)}, \end{aligned} \tag{5}$$

where

$$P^{(k)} = P_{i_1, i_2, \ldots, i_k}^{(k)} = g\left(\prod_{i=i_1}^{i_k} v_i x_i \right) \tag{6}$$

assuming

$$g(u) = u \text{ and } v_i = 1. \tag{7}$$

The higher-order terms can be again used as building blocks which are able to capture a high-order correlational

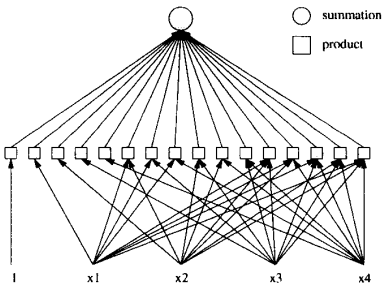

Figure 1: *An order 4 neuron*

structure of the data. In particular, by building a sigma unit which has as input various higher-order terms, we can construct a higher-order network of sigma-pi units:

$$y_i = f_i(u_i) = f_i\left(\sum_k w_k T^{(k)} \right) \tag{8}$$

The problem in using higher-order networks is that the number of terms explodes with the problem size; the number of parameters necessary for specifying an order k neuron is

$$r_k = \sum_{i=0}^{k} {}_nC_i, \tag{9}$$

because $w_{i_1 \cdots i_m}^{(m)}$ have ${}_nC_m$ components. Here n is the total number of inputs and ${}_nC_m$ are the binomial coefficients. As an example, an order 4 neuron has $2^4 = 16$ parameters as shown in Figure 1.

To avoid the combinatorial explosion, a method is needed to discover and combine useful terms and to eliminate non-essential terms. The following section describes a genetic programming method designed for these purposes.

III. Evolving Sigma-Pi Nets

In order to evolve problem-specific sigma-pi networks we use the breeder genetic programming (BGP) [35]. Similar to GPs, BGP uses tree representation for individuals. BGP uses, however, ranking-based truncation selection as in the breeder genetic algorithm (BGA) [26] instead of fitness-proportionate selection or tournament selection [2]. Another feature of BGP is its fitness function which uses the Occam's razor principle. The truncation selection combined with Occam's razor has been proved useful to balance the accuracy and parsimony of multilayer perceptrons.

The sigma-pi networks are represented as a set of m trees, where m is the number of output units. Each tree has an arbitrary number of subtrees. The grammar for describing the genotype is given in Figure 2. Figure 3 shows a genotype representation of a sigma-pi network depicted in Figure 4.

$$
\begin{array}{lcl}
\text{SPNN} & \longrightarrow & (\ Y_1\ Y_2\ \ldots\ Y_m\ ) \\
Y & \longrightarrow & (\ U\ r\ \theta\ W_1\ W_2\ \ldots\ W_r\ ) \\
U & \longrightarrow & \text{'S'} \mid \text{'P'} \\
W & \longrightarrow & (\ \text{'W'}\ w\ \{Y \mid X\}\ ) \\
X & \longrightarrow & \text{'X'}\ i \\
\theta & \longrightarrow & \Theta_{bin} \mid \Theta_{int} \\
\Theta_{bin} & \longrightarrow & -1 \mid +1 \\
\Theta_{int} & \longrightarrow & -r \mid \ldots \mid 0 \mid \ldots \mid +r \\
w & \longrightarrow & \Omega_{bin} \mid \Omega_{int} \\
\Omega_{bin} & \longrightarrow & -1 \mid +1 \\
\Omega_{int} & \longrightarrow & 0 \mid \pm 1 \mid \pm 2 \mid \pm 3 \mid \ldots \\
r & \longrightarrow & 1 \mid 2 \mid 3 \mid \ldots \\
i & \longrightarrow & 1 \mid 2 \mid 3 \mid \ldots \mid n
\end{array}
$$

Figure 2: *Grammar for generating the genotype of a feed-forward network of sigma-pi units. A network is represented as a set of m trees, each having an arbitrary number of subtrees. Each leaf of the trees indexes one of the n external input units.*

Each nonterminal node in the tree represents a sigma or pi unit. The symbol S is used for sigma units and the symbol P for pi units. Each nonterminal has $r+2$ arguments. These arguments denote r weights, $w_1, w_2, ..., w_r$, the threshold θ_i of the unit, and the receptive field size r itself. The integer r is chosen randomly and adapted during evolution. Each unit computes the weighted sum or product of r inputs, followed by a threshold function. The sigma unit computes

$$y_i = \begin{cases} +1 & \text{if } \sum_{j \in R(i)} w_{ij} y_j > \theta_i \\ -1 & \text{otherwise,} \end{cases} \qquad (10)$$

whereas a pi unit evaluates

$$y_i = \begin{cases} +1 & \text{if } \prod_{j \in R(i)} w_{ij} y_j > \theta_i \\ -1 & \text{otherwise.} \end{cases} \qquad (11)$$

In the genotype the weights are represented as separate nodes, denoted by the symbol W. A W-node has two arguments. The first argument denotes weight value w and the second points recursively to other nonterminal units (S or P) or an external input. The external input is represented by a symbol X followed by an input index. Thus the terminal set is $\{X1, X2, ..., Xn\}$, where n is the number of input units which is determined by the problem size. The weights and thresholds are allowed to be binary or integer values.

The top-level structure of the BGP algorithm is described in Figure 5. The initial population $\mathcal{A}(0)$ of M networks is generated at random. The random initialization includes the type and receptive field of units, the depth of the network, and the values of weights and thresholds. Then, for the gth generation the fitness values $F_i(g)$ of networks are evaluated using the training set of N examples. If the termination condition is satisfied, the algorithm stops. Otherwise, it selects the best τM networks of gth population into the mating pool $\mathcal{B}(g)$, where $\tau \in (0, 1]$ is the truncation threshold. Each network in $\mathcal{B}(g)$ undergoes a local

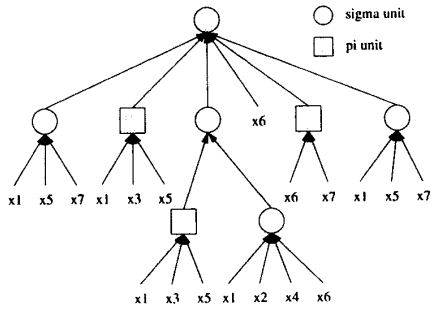

Figure 3: *Genotype of a sigma-pi network. Each nonterminal node is a sigma or pi unit having a receptive field of flexible size.*

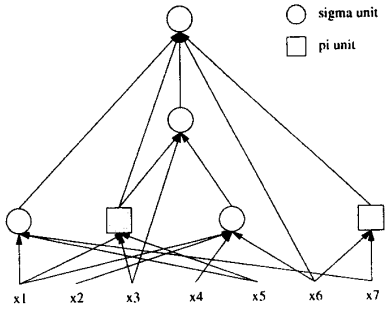

Figure 4: *Phenotype of a sigma-pi network. This architecture allows higher-order terms with local receptive fields. Direct connections between non-neighboring layers are also allowed.*

hillclimbing to be described later. This results in revised mating pool $\mathcal{B}(g)$. The $(g+1)$st generation of size M is produced by applying crossover operators to randomly chosen parent networks in the mating pool $\mathcal{B}(g)$. We use the elitist strategy by replacing the worst fit network $\mathcal{A}_{worst}(g+1)$ in $\mathcal{A}(g+1)$ by the best $\mathcal{A}_{best}(g)$ of $\mathcal{A}(g)$. A new population is generated repeatedly until an acceptable solution is found or the variance of the fitness $V(g)$ falls below a specified limit value V_{min}:

$$V(g) = \frac{1}{M} \sum_{i=1}^{M} \left(F_i(g) - \bar{F}(g)\right)^2 \leq V_{min} \qquad (12)$$

where $\bar{F}(g)$ is the average fitness of the individuals in $\mathcal{A}(g)$. The algorithm also stops if it reaches the maximum number of generations, g_{max}.

The weights of a network are trained by a local hillclimbing method to each selected parent. Many hillclimbing methods are possible. We use a simple next-ascent hillclimbing procedure: given an individual B_i, a better individual B_i^{new} is sought by repeatedly applying the mutation operator to the weights until there is no weight configuration found having better fitness in each sweep through the

```
procedure BGP(M, τ, V_min, g_max)
    population size: int M
    truncation rate: real τ
    variance: real V_min, V(g)
    generation: int g_max, g
    fitness values: real F_i
    population: array A = (A_1, A_2, ..., A_M)
    mating pool: array B = (B_1, B_2, ..., B_τ·M)

    g ← 0
    A(0) ← Initialize(M)
    while (V(g) > V_min) do
        F_i(g) ← Evaluate(A_i(g))    ∀i ∈ {1, ..., M}
        if (solution found or g ≥ g_max) stop
        B(g) ← Select(A(g), F(g), τ)
        B_i(g) ← Hillclimb(B_i(g))    ∀i ∈ {1, ..., τM}
        A(g + 1) ← Mate(B(g), M)
        A_worst(g + 1) ← A_best(g)
        g ← g + 1
    endwhile
endprocedure
```

Figure 5: *The top-level structure of the BGP algorithm*

individual. The sequence of mutation is defined as the depth-first search order.

The mutation operation is performed by replacing the value of a node, c_i, of the tree by another, i.e. by finding the class C_k of c_i and replacing c_i by another member $c_j, j \neq i$ in the set C_k. Here the class C_k must first be found because not every value (node) can be mutated to arbitrary values. For example, a weight value must be drawn from the set $\{+1, -1\}$. The thresholds are mutated the same way as the weights. The index for the input units can be mutated by another input index. We also allow a nonterminal symbol S to be mutated by a P and vise versa, i.e. changing the type of neural units. This flexibility ensures that multilayer perceptrons can also be evolved from sigma-pi networks.

The crossover operator adapts the size, depth and receptive field shape of the network architecture. The crossover operation starts with choosing randomly two parents, B_i and B_j, from the mating pool $\mathcal{B}(g)$. The nodes in the tree are numbered according to the depth-first search order and crossover sites c_i and c_j are chosen at random with the following conditions: $1 \leq c_i \leq Size(B_i)$ and $1 \leq c_j \leq Size(B_j)$. The length $Size(B_k)$ of an individual B_k is defined as the total number of units and weights. The subtrees of two parent individuals, B_i and B_j, are exchanged at the given crossover points to form two offspring B_i' and B_j'. The label of the nodes, c_i and c_j, must belong to the same class, i.e. either both U-type or both W-type nodes. The number of arguments of each operator plays no role because the syntactically correct subtree under the node is completely replaced by another syntactically correct subtree.

We use the following equation as the fitness measure

$$F(D|W,A) = \frac{E(D|W,A)}{m \cdot N} + \frac{C(W|A)}{N \cdot C_{max}}. \qquad (13)$$

A theoretical background behind this fitness function is discussed in [35]. The first term expresses the accuracy penalty caused by the error for the training set:

$$E(D|W,A) = \sum_{i=1}^{N} \sum_{j=1}^{m} (y_{ij} - o_j(x_i; W, A))^2 . \qquad (14)$$

Here y_{ij} denotes the jth component of the ith desired output vector y_i, and $o_j(x_i; W, A)$ denotes the jth actual output of the network with the architecture A and the set of weights W for the ith training input vector x_i.

The second term in the fitness function expresses the complexity penalty of the network, often called Occam's razor. The complexity is defined as

$$C(W|A) = W(A) + 10 \cdot L(A) + U(A), \qquad (15)$$

where $W(A) = \sum_{k=1}^{K} w_k^2$ is the number of weights in the network for binary weights. $L(A)$ and $U(A)$ denote the number of layers and units, respectively. The $L(A)$ term penalizes a deep architecture which requires a large execution time after training. The $U(A)$ term penalizes a large number of units whose realization is more expensive than weights. Notice that the $L(A)$ term is multiplied by 10 to penalize it more strongly than others. Depending on applications, one may weight three terms differently. C_{max} is a normalization factor used for the complexity term to be between 0 and 1. In all experiments we set $C_{max} = 1000$, assuming the problems can be solved by $C(W|A) \leq 1000$.

Notice in equation (13) that the complexity term $C(W|A)$ is divided by N, the number of training examples, to have the complexity term play a minor role in determining the total fitness value of the network. This ensures a small network be preferred to a large network only if both of them achieve a comparable performance. Otherwise, the evolution may not lead to a solution by preferring smaller networks which lack the capacity to learn the training set.

IV. Simulation Results

The method was tested on the parity problem. We performed two kinds of experiments separately. In the first, we are interested to know whether the use of product units is effective and, if yes, to what extent. In these experiments, noise-free examples are used. For the second series of experiments, we use noisy data. The generalization performance and the learning speed of different strategies are compared to study the effect of Occam's razor for the construction of sigma-pi networks.

A. Clean Data

The accuracy and convergence speed of sigma-pi networks are studied. We also measured the network complexity in terms of the number of layers, units and connections.

The results are summarized in Table 1. For these experiments we used noise-free data consisting of 2^n examples for problem size n. We performed 10 runs for each problem size. The population was initialized for every indidual 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, $M = 400$ and $g_{max} = 80$ for $n = 8$.

n	arch	layr	unit	conn	accr	gen
2	S	2.2	4.0	12.9	95.0	12.5
	S/P	2.0	2.2	6.5	100.0	2.9
4	S	4.9	17.9	55.0	81.9	40.0
	S/P	2.4	3.6	14.8	100.0	8.5
6	S	4.4	30.2	161.4	89.5	46.2
	S/P	2.9	8.9	40.2	98.4	35.8
8	S	5.9	65.6	414.1	86.4	63.4
	S/P	3.6	21.0	106.8	98.1	56.2

Table 1: *Results for parity problems*

To test the effectiveness of pi units we also run 10 experiments using only sigma units. The experiments were the same as before except that all units are initialized as sigma units. As the results show, the additional use of pi units consistently improved the performance in accuracy as well as in complexity reduction.

B. Noisy Data

In the second set of experiments we used 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. This means, on average, 6 or 7 examples out of 128 have false outputs. The generalization performance of the best solution in each generation was tested by the complete data set of $2^8 = 256$ noise-free examples.

Figure 6 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 7 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 with the change of its complexity.

The performance of the BGP with the fitness function (13) was compared with a method that uses just the error term as the fitness measure, i.e.

$$F(D|W,A) = \frac{E(D|W,A)}{m \cdot N}. \qquad (16)$$

Both methods used the same noisy data of the 8-parity problem. For each method, 10 runs were executed until the 80th generation to observe the training and generalization performance of the solutions.

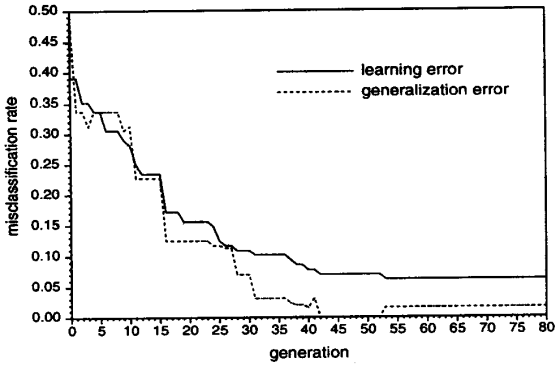

Figure 6: *The evolution of the network performance for noisy data of the 8-input parity function. Also shown is the generalization performance on the complete test set of noise-free examples.*

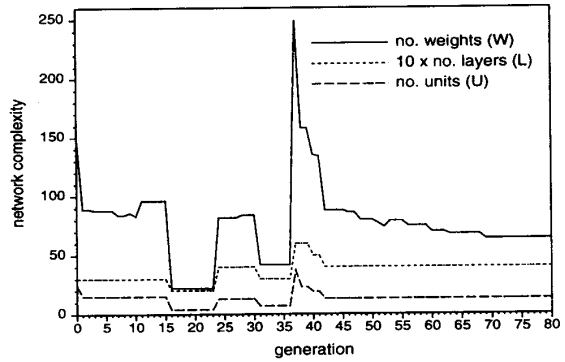

Figure 7: *The evolution of the network size for noisy data of the 8-input parity function.*

Table 2 shows the average network size found at the 80th generation. The corresponding performance and learning time are shown in the table. The learning time is measured in millions of evaluations of arithmetic operations associated with calculating activation values of neural units. The results show that applying Occam's razor achieves significantly better performance for this problem. Without Occam's razor the network size increased to an arbitrarily large size, which makes it difficult to find a useful building block to combine. Another advantage of using Occam's ra-

	F = E	F = E + C
number of layers	7.8 ± 0.1	3.5 ± 0.2
number of units	65.4 ± 1.8	19.0 ± 2.7
number of weights	391.8 ± 23.1	103.8 ± 19.2
learning accuracy	75.1 ± 2.5	89.3 ± 1.8
generalization	61.2 ± 0.8	89.9 ± 3.8
num. evaluations	2630.4 ± 136.1	1432.0 ± 112.2

Table 2: *Performance with and without Occam's razor*

zor is the accelerated convergence. In these experiments, the proposed fitness function decreased the network size by approximately four times and the speed-up factor of learning was two.

V. Concluding Remarks

Necessity and usefulness of higher-order neural networks have been well-known. However the explosively increasing number of terms has hampered the design and training of higher-order networks. The present work shows the potential effectiveness of genetic programming to handle this problem. In particular, we show how the sigma-pi neural networks can represent the higher-order terms and how BGP can be extended to synthesize problem-specific sigma-pi networks. We also confirm the usefulness of Occam's razor in genetic programming for the improvement of generalization and convergence speed. Experimental results show that sigma-pi networks solve parity-like problems significantly better than perceptron networks. In another set of experiments, we found that the method can also evolve usual multilayer perceptrons with slightly more costs when the product units are unnecessary for solving the problem.

Acknowledgement

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

References

[1] S. Amari, "Dualistic geometry of the manifold of higher-order neurons," *Neural Networks*, vol. 4, pp. 443–451, 1991.

[2] P. J. Angeline and J. B. Pollack, "Competitive environments evolve better solutions for complex tasks," in *Proc. Fifth Int. Conf. Genetic Algorithms (ICGA-93)*, Morgan Kaufmann, 1993, pp. 264–270.

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

[4] R. J. Collins, *Studies in Artificial Evolution*, Ph.D. thesis, University of California, Los Angeles, 1992.

[5] R. Durbin and D. E. Rumelhart, "Product units: A computationally powerful and biologically plausible extension to backpropagation networks," *Neural Computation*, vol. 1, pp. 133–142, 1989.

[6] J. A. Feldman and D. H. Ballard, "Connectionist models and their properties," *Cognitive Science*, vol. 6, pp. 205-254, 1982.

[7] J. Franke, "On the functional classifier," *Proc. First Int. Conf. Document Analysis and Recognition*, 1991.

[8] C. L. Giles and T. Maxwell, "Learning, invariance, and generalization in high-order neural networks," *Applied Optics*, vol. 26, no. 23, pp. 4972–4978, 1987.

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

[10] F. Gruau, "Genetic synthesis of Boolean neural networks with a cell rewriting developmental process," Tech. Rep., Laboratoire de l'Informatique du Parallélisme, 1992.

[11] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," in *Proc. Third Int. Conf. Genetic Algorithms (ICGA-89)*, D. Schaffer, Ed. Morgan Kaufmann, pp. 360–369.

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

[13] H. Iba, T. Kurita, H. de Garis, and T. Sato, "System identification using structured genetic algorithm," in *Proc. Fifth Int. Conf. Genetic Algorithms (ICGA-93)*, S. Forrest, Ed. Morgan Kaufmann, 1993, pp. 279–286.

[14] A. G. Ivakhnenko, "Polynomial theory of complex systems," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 1, pp. 364–378, 1971.

[15] H. Kargupta and R. E. Smith, "System identification with evolving polynomial networks," in *Proc. Fourth Int. Conf. Genetic Algorithms (ICGA-91)*, R. Belew and L. Booker, Ed. Morgan Kaufmann, 1991, pp. 370–376.

[16] K. E. Kinnear Jr., "Generality and difficulty in genetic programming: Evolving a sort," in *Proc. Fifth Int. Conf. Genetic Algorithms (ICGA-93)*,Morgan Kaufmann, 1993, pp. 287–294.

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

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

[19] J. R. Koza and J. P. Rice, *Genetic Programming: The Movie*. MIT Press, 1992.

[20] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer, 1992.

[21] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proc. Third Int. Conf. Genetic Algorithms (ICGA-89)*, Morgan Kaufmann, 1989, pp. 379–384.

[22] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969, 1988.

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

[24] H. Mühlenbein, "Parallel genetic algorithms in combinatorial optimization," in *Computer Science and Operations Research*, G. Balci, R. Sharda, and S. A. Zenios, Ed. Pergamon, 1992, pp. 441–456.

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

[26] H. Mühlenbein and D. Schierkamp-Voosen, "Predictive models for the breeder genetic algorithm I: Continuous parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 25–49, 1993.

[27] D. E. Rumelhart, G. E. Hinton, and J. L. McClelland, "A general framework for parallel distributed processing," in *Parallel Distributed Processing*, Vol. I, D. E. Rumelhart and J. L. McClelland, Ed. MIT Press, 1986, pp. 45–76,

[28] R. Sorkin, "A quantitative Occam's razor," *International Journal of Theoretical Physics*, vol. 22, pp. 1091–1104, 1983.

[29] W. A. Tackett, "Genetic programming for feature discovery and image discrimination," in *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA-93)*, S. Forrest, Ed. Morgan Kaufmann, 1993, pp. 303–309.

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

[31] B. T. Zhang, *Learning by Genetic Neural Evolution*. Ph.D. thesis in German, ISBN 3-929037-16-5, Sankt Augustin: Infix-Verlag, 1992. Also available as Informatik Berichte No. 93, Institut für Informatik I, Universität Bonn, D-53117 Bonn, 1992.

[32] B. T. Zhang, "Accelerated learning by active example selection," to appear in *International Journal of Neural Systems*, 1993.

[33] B. T. Zhang, "Teaching neural networks by genetic exploration," *GMD-Arbeitspapiere*, No. 805, German National Research Center for Computer Science, 1993.

[34] B. T. Zhang and H. Mühlenbein, "Genetic programming of minimal neural nets using Occam's razor," in *Proc. Fifth Int. Conf. Genetic Algorithms (ICGA-93)*, S. Forrest, Ed. Morgan Kaufmann, 1993, pp. 342–349.

[35] B. T. Zhang and H. Mühlenbein, "Evolving optimal neural networks using genetic algorithms with Occam's razor," to appear in *Complex Systems*, 1993.

[36] R. Zembowicz and J. M. Żytkow, "Discovery of equations: Experimental evaluation of convergence," in *Proc. of AAAI-92*, Menro Park, CA: AAAI Press, 1992, pp. 70–75.