Proc. of 2nd European Congress on Intelligent Techniques and Soft Computing (EUFIT'94-Aachen), pp. 1265-1269, 1994

10

Genetic Breeding of Novel Neural Architectures

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 are computational techniques for search, optimization and machine learning that are inspired by the process of natural evolution. Genetic algorithms have been used for neural networks in several ways: to optimize the network architecture and to train the weights of a fixed architecture among others. While most previous work focuses on only one of these two options, this paper describes an alternative evolutionary computational approach in which the architecture and the weights are optimized simultaneously. As opposed to conventional learning algorithms for neural networks, the presented method makes relatively few assumptions about the network types. Thus the same method can also be used to breed other novel network architectures as well as standard ones. We demonstrate the effectiveness of the method on the synthesis of sigma-pi neural networks.

1 Introduction

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 [1, 7]. 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.

Both genetic algorithms and neural networks are computational paradigms that, while not actulally 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. Each of these paradigms has been found useful in solving certain engineering problems [11].

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 away from fully understanding biological neural systems, we are also still far away from fully understanding biological genetic codes. Various schemes for combining genetic algorithms and neural networks have been proposed and tested in recent years [11, 12]. They include

1. selecting features for use by neural network classifiers. This combination has already achieved some success on real world tasks.
2. determination of neural network weights. Where gradient or error information is not available, genetic algorithms may be a promising training method.
3. optimization of the network topology. Several methods have been proposed for evolving network topologies as will be described below.

In section 2 we briefly survey some of the encoding schemes and present a new method for representing and evolving general feedforward network architectures. The simulation results are given in section 3 and discussed in section 4.

2 Genetic Encoding of Neural Nets

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 [6]. 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 [9]. Recent works, however, have focused on using genetic algorithms separately in each optimization problem, mainly in optimizing the network topology. Harp *et al.* [2] and Miller *et al.* [5] 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.* [12] to prune unnecessary connections. Kitano [3] 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 [10], a gradient-descent method, to train the weights of the network. Koza [4] provides an alternative approach to representing neural networks, under the framework of so-called genetic programming (GP), which enables modification not only of the weights but also of the architecture for a neural network. However, this method does not provide a general method for representing an arbitrary structure.

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:

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

In this tree representation, 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, $k \in \{1, \cdots, n\}$.

This encoding scheme can represent any feedforward network with local receptive fields and direct connections between non-neighboring layers (see [14] for more details). This is contrasted with the more commonly used perceptron architecture of fully connected feedforward networks. Figure 1 shows the corresponding tree representation.

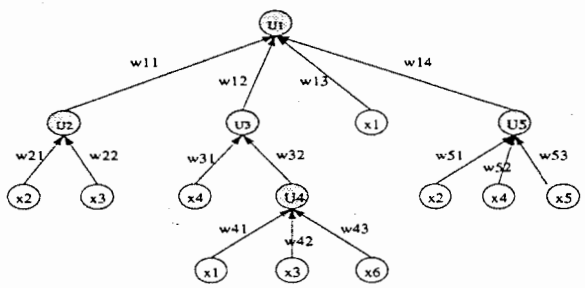

Figure 1: *Tree representation of the network. For simplicity, the thresholds are not shown.*

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, $u_i = \sum_{j \in R(i)} w_{ij} x_j$, a pi-unit builds a *product* of weighted inputs: $u_i = \prod_{j \in R(i)} w_{ij} x_j$. 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

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

which require a number of layers consisting of summation units in conventional neural networks. 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 network of sigma-pi units can be constructed:

$$y_i = f_i(u_i) = f_i\left(\sum_k w_{ik} T^{(k)}\right) = f_i\left(\sum_k w_{ik} f_k\left(\prod_{j=1}^{j_k} w_{kj} x_j\right)\right). \quad (2)$$

In order to evolve the architectures we use breeder genetic programming (BGP) [13, 14]. As discussed above, BGP uses tree representation for individuals, a feature derived from GP. BGP uses, however, ranking-based truncation selection as in the breeder genetic algorithm (BGA) [8] instead of fitness-proportionate selection. Another feature of BGP is its fitness function which uses the Occam's razor principle. Occam's razor states that unnecessarily complex models should not be preferred to simpler ones. The truncation selection combined with Occam's razor has been proved useful in balancing the accuracy and parsimony of multilayer perceptrons.

We maintain a population $\mathcal{A}$ consisting of M individuals A_i of variable size. Each individual represents a neural network. 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 satisfied, then stop the evolution. Otherwise, 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.

The networks of the initial population, $\mathcal{A}(0)$, are generated with a random number of layers. The receptive field of each neural unit and its width are also chosen randomly. The $(g+1)$th population, $\mathcal{A}(g+1)$, is created from $\mathcal{A}(g)$ in three steps: selection, hillclimbing, and mating. In the selection step, the fittest τM individuals in $\mathcal{A}(g)$ are accepted into the mating pool $\mathcal{B}(g)$. The parameter τ determines the selection intensity and has a value from the interval $(0, 1]$. After selection, each individual in $\mathcal{B}(g)$ undergoes a hillclimbing search where the weights of the network are adapted. While a simple hillclimbing method is sufficient for adjustment of discrete weights, other search methods might as well been used for this purpose. For real-valued weights, it will be necessary to modify or replace the discrete hillclimbing search by a continuous parameter optimization method which may be again genetic algorithms [7].

Hillclimbing results in the revised mate set $\mathcal{B}(g)$. The mating phase repeatedly selects two random parent individuals in $\mathcal{B}(g)$ to produce two offspring in the new population $\mathcal{A}(g+1)$ by applying crossover operators, until the population size amounts to M. The crossover operator adapts the size, depth and receptive field shape of the network architecture. The crossover operation chooses randomly two parents, B_i and B_j, from the mating pool $\mathcal{B}(g)$. The subtrees of two parent individuals, B_i and B_j, are then exchanged at the given crossover points to form two offspring B_i' and B_j'. 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. The mutation operation may be applied to change some value of a node. 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}. The algorithm also stops if a specified number of generations, g_{max}, is carried out.

We use a training set D to measure the fitness F_i of the network A_i with weights W:

$$F_i = F(D|W, A_i) = \frac{E(D|W, A_i)}{m \cdot N} + \frac{C(W|A_i)}{N \cdot C_{max}}, \quad (3)$$

where N is the number of training examples. A theoretical background behind this fitness function is discussed in [14]. 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.

3 Experimental Results: Evolving Sigma-Pi Networks

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 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%.

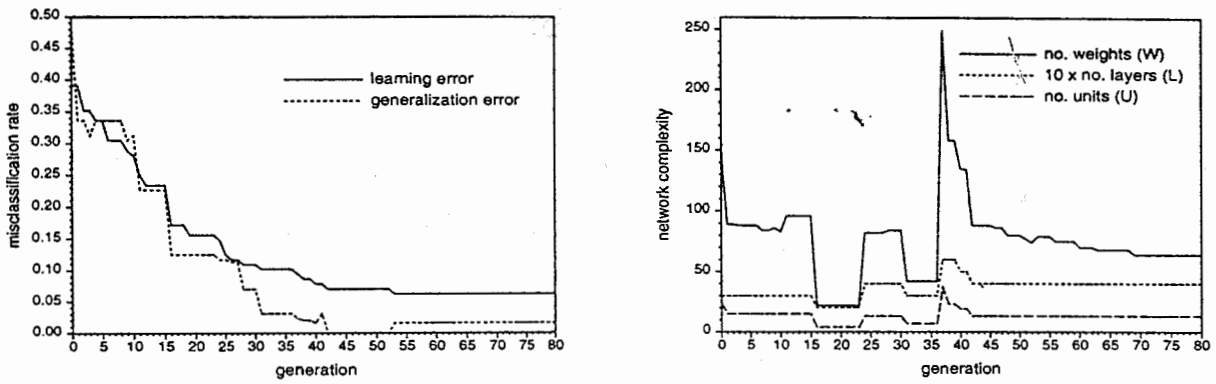

Figure 2: *Evolving sigma-pi networks from noisy data*

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 with the change of its complexity.

To see the effect of Occam's razor, we performed experiments with the fitness function without the complexity term, i.e. $F(D|W, A_i) = E(D|W, A_i)/mN$. The results showed 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 razor is the accelerated convergence. The proposed fitness function decreased the network size by approximately four times and the speed-up factor of learning was two.

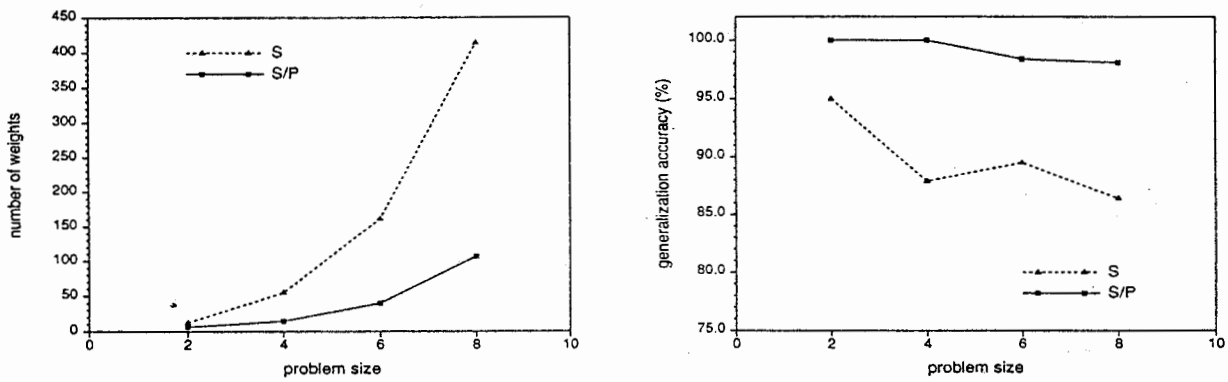

Figure 3: *Performance comparison of perceptrons and sigma-pi networks*

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 $n = 2, 4, 6, 8$. We performed 10 runs for each problem size. 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 run 10 experiments using only sigma units. The experiments were the same as before except that all units are initialized as sigma units. The comparison results are summarized in Figure 3. As the graphs show, the additional use of pi units consistently improved the performance in accuracy as well as in complexity reduction. This implies that in solving parity problems, the pi units were more effective than the sigma units and that the BGP could construct such an architecture.

4 Concluding Remarks

We have presented an evolutionary method for constructing novel 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. In contrast to conventional learning algorithms for neural networks, the presented method makes relatively few assumptions about the network types. Thus the same method can be used to breed non-standard type neural architectures as well as the standard ones. 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. The potential for evolving novel neural architectures that are customized for specific applications is one of the most interesting properties of genetic algorithms.

References

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

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

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

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

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

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

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

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

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

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

[11] 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*, 1–37 (IEEE, 1992).

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

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

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

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

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