Published in *Evolutionary Computation*, Vol. 5, No. 2, pp. 213-236, 1997.

Evolutionary Induction of Sparse Neural Trees

Byoung-Tak Zhang
Dept of Computer Engineering
Seoul National University
Seoul 151-742, Korea
E-mail: btzhang@comp.snu.ac.kr

Peter Ohm
German National Research Center for
Information Technology (GMD)
D-53754 St. Augustin, Germany
E-mail: peter.ohm@gmd.de

Heinz Mühlenbein
German National Research Center for
Information Technology (GMD)
D-53754 St. Augustin, Germany
E-mail: heinz.muehlenbein@gmd.de

Abstract

This paper is concerned with the automatic induction of parsimonious neural networks. In contrast to other program induction situations, network induction entails parametric learning as well as structural adaptation. We present a novel representation scheme, called neural trees, that allows efficient learning of network architectures and parameters both by genetic search. A hybrid evolutionary method is developed for neural tree induction that combines genetic programming and the breeder genetic algorithm under the unified framework of the minimum description length principle. The method is successfully applied to the induction of higher-order neural trees, while still keeping the resulting structures sparse to ensure good generalization performance. Empirical results are provided on two chaotic time-series prediction problems of practical interest.

Keywords: Program induction, genetic programming, higher-order neural networks, neural tree representation, minimum description length principle, time series prediction, breeder genetic algorithm.

1 Introduction

Higher-order neural networks are a generalization of the multilayer perceptrons, where the presence or absence of nonlinear interactions between different inputs are directly modeled in the network topologies (Kowalczyk and Ferra, 1994). Higher-order neural networks provide inherently more powerful learning capabilities than standard multilayer perceptrons (Fahner and Eckmiller, 1994). In spite of these potential benefits, the higher-order neural networks are practically limited by the combinatorial explosion of higher-order terms with increasing problem size. A closely related disadvantage is that complete higher-order neurons do not produce good generalization.

Most of current methods for avoiding combinatorial explosion rely on *a priori* knowledge about the problem to generate and preselect useful higher-order terms (Kowalczyk and Ferra, 1994). This inherently limits the number of applications to which higher-order networks are applied, as in most neural network applications little knowledge exists about the solution to the problem in question. Another class of algorithms attempt to avoid combinatorial explosion by setting a bound on the highest order (Giles and Maxwell, 1987). There are, however, many important classes of functions that are of high or even unlimited order (Minsky and Papert, 1969). These algorithms tend to force a task into an assumed architecture class rather than fitting an appropriate architecture to the task.

The present work takes the opposite approach: we want to explore a more general class of higher-order architectures, while keeping the resulting structures sparse to promote good generalization. This is an inductive learning problem in which both the model and its parameters should be fitted to the data. The problem matches well with the genetic programming (GP) paradigm (Koza, 1992) in that both the size and shape of structures are to be evolved. On the other hand, this problem is contrasted with typical GP applications where evolved programs contain only a small number of numerical components.

The search space for neural network induction consists of two levels. One is the space of all possible network architectures (models). The other is the space of all possible weight configurations for a given architecture (parameters). The difficulty here is that an architecture cannot be evaluated without the assignment of weights and likewise a weight vector cannot be evaluated without knowing the underlying network architecture. Both optimizations must be interleaved, as was done in the GNARL system (Angeline, Saunders and Pollack, 1994) which constructs recurrent neural networks using evolutionary programming. However, relative few GP applications have analyzed the interaction between structural adaptation and parametric learning.

In this paper we present a hybrid evolutionary method in which the neural architectures are evolved by genetic programming while the parameters are fitted by a local search based on the breeder genetic algorithm (BGA), a real-valued genetic algorithm designed for continuous parameter optimization (Mühlenbein and Schlierkamp-Voosen, 1994). The crux of the method is the neural tree representation, a tree representation of a neural network with various neuron types. This representation scheme allows efficient adaptation of network architectures and parameters both by simple genetic operators. Neural trees can represent a wide range of nonlinear mappings. They do not require an explicit decoding process, allowing training and evaluation to be performed directly with genotype. The subtree crossover developed for genetic programming naturally applies to this representation. Neural trees are also very convenient for evolutionary optimization of weight values.

A minimum description length (MDL)-based fitness function is used to achieve parsimonious programs without sacrificing the training accuracy of the evolved programs. Parsimony is important to minimize the variance error and thus to improve the generalization accuracy of the programs (Zhang and Mühlenbein, 1995; Zhang and Mühlenbein, 1996). Other techniques developed for improving the efficiency of the entire induction process include libraries of building blocks, local fitness-based crossover, injection and pruning of submodules, and scheduling of genetic operators. The investigated method is successfully applied to the induction of a broad class of higher-order neural networks with unlimited order, while still keeping the resulting structures sparse. Empirical results are provided on two benchmarking time-series prediction problems, namely the Mackey-Glass equation and chaotic fluctuations in a far-infrared laser.

The paper is organized as follows. Section 2 gives background on higher-order neurons. Section 3 presents the neural tree representation scheme. Section 4 describes the evolutionary algorithms for adapting the structures and weights of neural trees. Section 5 provides the experimental results and analyzes the characteristics of the proposed method. Section 6 contains conclusions. A theoretical analysis of computational complexity for the neural tree induction problem is given in Appendix A.

2 Higher-Order Neurons

Higher-order neural networks are a generalization of the multilayer perceptrons (Minsky and Papert, 1969), where the presence or absence, order, and strengths of nonlinear interactions between different inputs are directly modeled and thus closely mirrored in the network topologies. Figure 1 illustrates the structures of a multilayer perceptron (MLP) and a higher-order neural network (HONN). MLPs employee one or more layers of the summation (or sigma) units which compute a linear sum of weighted inputs:

$$net_i(\mathbf{x}) = w_{i0} + \sum_j w_{ij} x_j \tag{1}$$

where w_{i0} is the bias term and x_j are the inputs to the ith neuron. The higher-order interactions are implicitly represented in MLPs by a cascade of several layers of sigma units. In contrast, HONNs make explicit use of higher-order terms or pi (product) units. In case of the HONN depicted in Figure 1 (right), each of the order-6 neurons at the output layer computes:

$$\begin{aligned} net_i(\mathbf{x}) &= w_{i0} + \sum_j w_{ij}^{(1)} x_j + \sum_{j,k} w_{ijk}^{(2)} x_j x_k + \cdots \\ &\quad + \sum_{j,k,l,m,n,o} w_{ijklmno}^{(6)} x_j x_k x_l x_m x_n x_o \end{aligned} \tag{2}$$

where $w^{(s)}$ indicates a connection weight of sth order. For example, the second-order term is expanded as follows:

$$\begin{aligned} \sum_{j,k} w_{ijk}^{(2)} x_j x_k &= w_{212}^{(2)} x_1 x_2 + w_{213}^{(2)} x_1 x_3 + w_{214}^{(2)} x_1 x_4 + w_{215}^{(2)} x_1 x_5 + w_{216}^{(2)} x_1 x_6 \\ &\quad + w_{223}^{(2)} x_2 x_3 + w_{224}^{(2)} x_2 x_4 + w_{225}^{(2)} x_2 x_5 + w_{226}^{(2)} x_2 x_6 + w_{234}^{(2)} x_3 x_4 \\ &\quad + w_{235}^{(2)} x_3 x_5 + w_{236}^{(2)} x_3 x_6 + w_{245}^{(2)} x_4 x_5 + w_{246}^{(2)} x_4 x_6 + w_{256}^{(2)} x_5 x_6 \end{aligned} \tag{3}$$

In general, the net input of the complete higher-order neuron is given by the multinomial expansion

$$net_i(\mathbf{x}) = \sum_{\mathbf{a} \in \{0,1\}^n} w_{\mathbf{a}} \pi_{\mathbf{a}}(\mathbf{x}) \quad = \sum_{\mathbf{a} \in \{0,1\}^n} w_{\mathbf{a}} \prod_{j=1}^{n} x_j^{a_j} \tag{4}$$

where $w_{\mathbf{a}} \in \mathbb{R}$, $\mathbf{a} = a_1 ... a_n \in \{0,1\}^n$, and $\mathbf{x}$ denotes the original input vector. The 2^n monomials $\pi_{\mathbf{a}}$ are referred to as Walsh functions (Goldberg, 1989). They are orthogonal and span the space of real valued functions defined over binary strings. The expansion coefficients $w_{00..00}, w_{00..01}, ..., w_{11..11}$

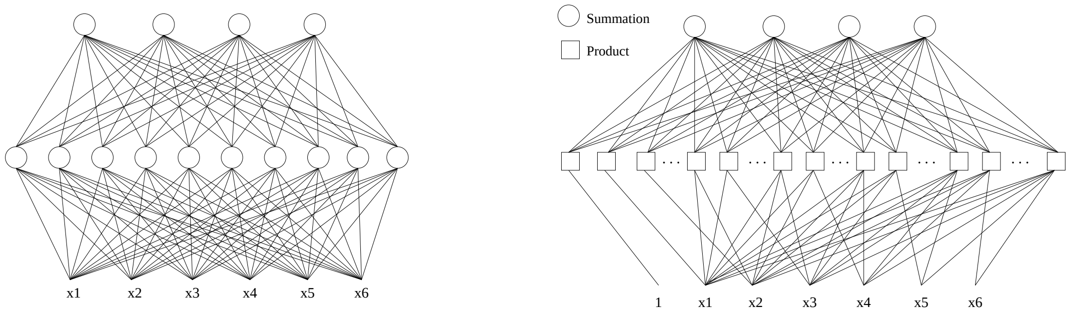


Figure 1: *The structure of a multilayer perceptron and a higher-order neural network in comparison. (left) Multilayer, fully-connected, and strictly layered MLP. (right) Single layer, fully-connected HONN.*

can be interpreted as synaptic weights of orders $\sum_{j=1}^{n} a_j$. Note that a complete higher-order neuron can be represented by a tree consisting of a summation (sigma) unit and 2^n product (pi) units.

The complete higher-order neuron is practically limited by the combinatorial explosion of higher-order terms with increasing number of inputs. The number of parameters necessary for specifying an order k neuron is $r_k = \sum_{i=0}^{k} {}_nC_i$, where n is the number of inputs. A closely related disadvantage is that it does not produce a generalization; it makes up a fast distributed memory with a target fixed at each corner of the hypercube (Fahner and Eckmiller, 1994).

One way to achieve sensible generalization, and to avoid the problem of combinatorial explosion at the same time, is to cut off all terms in expansion that exceed a certain order. For instance, the GMDH algorithm (Ivanknenko, 1971) uses as the basis function the polynomials of second degree:

$$net_i(\mathbf{x}) = G_i(\mathbf{x}) = w_{i0} + w_{i1}x_j + w_{i2}x_k + w_{i3}x_jx_k + w_{i4}x_j^2 + w_{i5}x_k^2 \tag{5}$$

where x_j and x_k are elements of input vector $\mathbf{x} = (x_1, x_2, ..., x_n)$. This seems a successful strategy especially for problems with a limited degree of nonlinearity, and helps to avoid overfitting when noise is present. However, if the order of the problem exceeds the allowed order, cascade of terms in multilayer networks is necessary. Figure 2 (left) illustrates an order-restricted network having a strictly-layered, and fully-connected regular architecture.

In this work we follow a more general approach that makes minimal architectural constraints to explore a broad class of higher-order networks. An example of the structure evolved in our framework is shown in Figure 2 (right). There is no explicit limit on the order of weights. Thus,

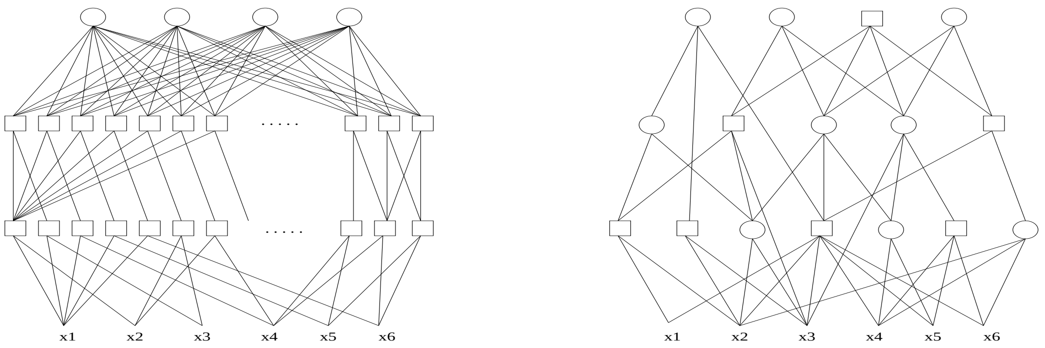


Figure 2: *An order-restricted neural network and an unlimited-order sparse network. (left) Multilayer, hierarchical, fully-connected, regular, and strictly layered HONN. (right) Multilayer, hierarchical, partially-connected, irregular, and non-strictly layered sparse HONN.*

interactions of arbitrary orders computed by the ultimate higher-order neuron can be realized within our framework. Cascading of higher-order terms in multilayers is permitted. No bound is enforced in the number of layers of the network. Instead, the overall network size is controlled implicitly by a complexity penalty imposed in the fitness function. The network structures are not strictly layered, i.e. each layer can have a mixture of sigma and pi units and connections between non-neighboring layers are allowed. The network may contain partial connectivity, which is useful for the economic representation of arbitrary higher-order interactions.

3 Neural Tree Encoding

The question of how the possible programs are represented as a genotype is critical to the working of an evolutionary approach to program induction. The representation or encoding used not only determines the class of programs that could possibly evolve but also constrain the choice of the evaluation process and thus efficiency of problem solving.

3.1 Related Work

Existing representation methods for neural networks can be roughly divided into two categories: direct and indirect encoding (Balakrishnan and Honavar, 1995). Direct encodings use a fixed structure, such as connection matrix or bitstrings that precisely specifies the architecture of the corresponding neural network. This encoding scheme requires little effort to decode. However, matrix structures have limited flexibility in expressing topologies of the network structure with variable layers. Bitstrings are not flexible enough to represent various partial connectivity without further annotation. Genetic operators need to be applied carefully to preserve the topological constraints of networks (Belew, McInerney and Schraudolph, 1992).

Indirect encoding schemes use rewrite rules to specify a set of construction rules that are recursively applied to yield the phenotype. Examples include graph generation grammars (Kitano, 1990) and cellular encoding (Gruau, 1993). This approach is interesting in that it simulates in some sense the developmental process. Subtree crossover applies well to these representations. In addition, experimental evidence has shown that the cellular encoding scheme is effective in evolving modular structures consisting of similar substructures (Gruau, 1993). However, the grammatical encoding does not seem appropriate for exploring a huge number of partial interaction possibilities as required in our application. In addition, grammatical encoding requires execution of rewrite-rules for every conversion from genotype to phenotype. This makes network training an expensive phase since training of neural networks requires a large number of evaluations and each evaluation needs a separate decoding.

The following subsection suggests an alternative representation which combines the advantages of direct and indirect encoding schemes. It is powerful and flexible in expressing a broad class of feedforward architectures. It is decoding-efficient and convenient for genetic operations.

3.2 Neural Trees

Let $\mathcal{NT}(d, b)$ denote the set of all possible trees of maximum depth d and maximum b branches for each node. The nonterminal nodes represent neural units and the neuron type is an element of the basis function set $\mathcal{F} = \{\text{neuron types}\}$. Each terminal node is labeled with an element from the terminal set $\mathcal{T} = \{x_1, x_2, ..., x_n\}$, where x_i is the ith component of the external input $\mathbf{x}$. Each link (j, i) represents a directed connection from node j to node i and is associated with a value w_{ij}, called the synaptic weight. The members of $\mathcal{NT}(d, b)$ are referred to as neural trees. In case of $\mathcal{F} = \{\Sigma, \Pi\}$, the trees are specifically called sigma-pi neural trees (Zhang and Muehlenbein, 1994). The root node is also called the output unit and the terminal nodes are called input units. Nodes that are not input or output units are hidden units. The layer of a node is defined as the longest path length to any terminal node of its subtree. The size of search space for the induction of neural trees is calculated in Appendix A.

Different neuron types are distinguished in the way of computing net inputs. For the evolution of higher-order networks, we consider two types of units. Sigma units compute the sum of weighted inputs from the lower layer:

$$net_i = \sum_j w_{ij} y_j \tag{6}$$

where y_j are the inputs to the ith neuron. Pi units compute the product of weighted inputs from the lower layer:

$$net_i = \prod_j w_{ij} y_j \tag{7}$$

where y_j are the inputs to i. The output of a neuron is computed either by the threshold response function

$$y_i = \sigma(net_i) = \begin{cases} 1 & : \quad net_i \geq 0 \\ -1 & : \quad net_i < 0 \end{cases} \tag{8}$$

or the sigmoid transfer function

$$y_i = f(net_i) = \frac{1}{1 + e^{-net_i}} \tag{9}$$

where net_i is the net input to the unit computed by Eqn. 6 or 7.

A higher-order network with m output units can be represented by m sigma-pi neural trees. That is, the genotype A_i of ith individual in our evolutionary framework consists of m neural trees:

$$A_i = (A_{i,1}, A_{i,2}, ..., A_{i,m}) \quad \forall k \in \{1, ..., m\} \; A_{i,k} \in \mathcal{NT}(d, b) \tag{10}$$

The population $\mathcal{A}(g)$ of generation g is the collection of neural networks in their tree representation:

$$\mathcal{A}(g) = (A_1, A_2, ..., A_M) \tag{11}$$

where M is the size of the population.

The neural tree representation does not restrict the functionality since any feedforward network can be represented with a forest of neural trees (Figure 3). The connections between input units to arbitrary units in the network is also possible since input units can appear more than once in the neural tree representation. The output of one unit can be used as input to more than one unit. In implementation, we have used the concept of library in which frequently-used fit submodules

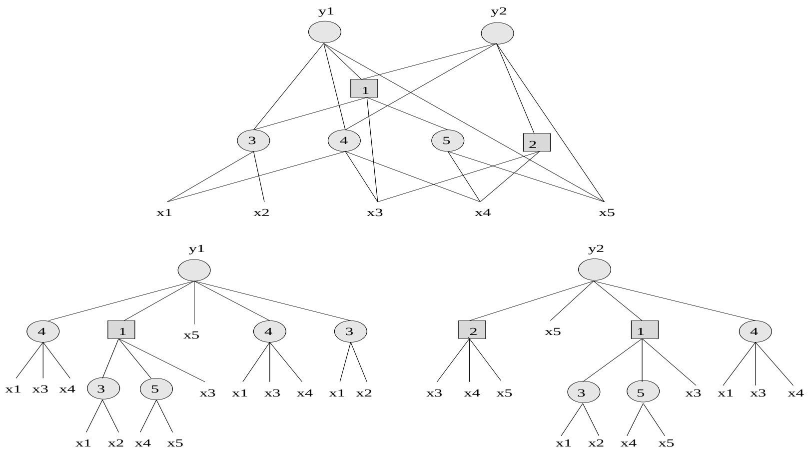


Figure 3: *Genotype (bottom) and its phenotype (top) of a sparse higher-order neural network.*

are stored as hidden units and multiply reused (see Section 4.2 below for more detailed description). This leads to the construction of modular structures and reduces memory requirements for representing the population.

Neural trees do not require decoding for their fitness evaluation. Training and evaluation of fitness can be performed directly on the genotype since both the genotype and phenotype are equivalent. Since subtree crossover used in genetic programming (Koza, 1992) applies without modification to this representation, we can use genetic programming as the main evolutionary engine.

4 Evolving Higher-Order Neural Trees

4.1 Initialization, Selection, and Termination

The main elements in using genetic programming to evolve a population of programs (in this case, the neural tree programs) for solving a given problem are: (1) the set of program primitives, namely the set of terminals $\mathcal{T}$ and the function set $\mathcal{F}$, (2) fitness measure F, (3) the control parameters, such as population size M and maximum number of generations g_{max}, and (4) termination criterion. In evolving the higher-order networks, the function set is defined as $\mathcal{F} = \{\Sigma, \Pi\}$. Though this function set contains only two elements, a realization of it is more complex: a nonterminal node

u_i is instantiated with the following structure:

$$u_i(w_{i0}, w_{i1}, w_{i2}, ..., w_{ib_i}) \quad \text{with} \quad u_i \in \mathcal{F} \tag{12}$$

where b_i denotes the number of inputs of the ith unit, w_{i0} is the bias, and $w_{ij}, j \neq 0$ are connection weights from unit j to i. The terminal set is defined as $\mathcal{T} = \{x_1, x_2, ..., x_n\}$, where n is the number of external inputs.

The population is initialized with randomly generated neural trees. The number of layers is chosen from a uniform distribution over $[1, d_{max}]$, where d_{max} is the user-specified value. Each of the nonterminal and terminal nodes is randomly assigned a label from the function set $\mathcal{F}$ and the terminal set $\mathcal{T}$, respectively. The number of inputs b_i for each neuron is chosen from a uniform distribution over $[1, b_{max}]$, where b_{max} is a user-supplied value. Typical values of d_{max} and b_{max} set at simulations are $d_{max} = 3$ and $b_{max} = n$, where n is the number of external inputs. The weights are initialized with random values selected uniformly from the range $[-1, 1]$. Though not necessary, some background knowledge on the problem, if available, may help improve the efficiency of evolution and can be exploited in the initialization phase. This knowledge includes labeling of unit types, branching factors, and tree depth.

In each generation, the trees are evaluated by a fitness function (defined in Section 4.4 below). We use an elitist truncation selection strategy, as in BGA (Mühlenbein and Schlierkamp-Voosen, 1994) and GNARL (Angeline, Saunders and Pollack, 1994). Top τ% of best individuals are selected as parents, from which M individuals of the next generation are produced. The best individual so far is kept by replacing the worst individual in the current population. Typical value of τ in the simulation studies was $\tau = 50\%$. Generating an offspring involves three steps: crossover, mutation, and local search. Crossover and mutation modifies the structural features of the networks and local search makes a fine-tuning of weights for a fixed architecture. New populations are iteratively generated until the following condition is met:

$$(F_{best}(g) \leq F_{min}) \text{ or } (Var(g) \leq Var_{min}) \text{ or } (g > g_{max}). \tag{13}$$

The first condition says that the evolution stops if the fitness value of the best individual in the population $F_{best}(g) = F(A_{best}(g))$ reaches the prespecified value F_{min}. The second temination case is when the variance of population fitness $Var(g) = Var(\mathcal{A}(g))$ goes down to the prespecified value Var_{min}. Finally, the process terminates if the generation number g exceeds the specificed maximum g_{max}.

4.2 GP-Based Structural Adaptation

Structural adaptations alter the topology, size, depth, and shape of neural networks: the first three are changed by crossover and the last by mutations.

As with other genetic programming methods, crossover is performed by exchanging subtrees of parent trees. Because of the closure property of tree representations (Koza, 1992), no syntactic restriction is necessary in choosing the crossover points. Instead of producing two offspring, we create only one, which allows a guided crossover by subtree evaluation.

Several criteria for subtree evaluation have been proposed in the literature, including the error of the substree, error difference, frequency of subtrees, use of the average fitness of population, correlation-based selection, combination of frequency and error difference (see (Rosca, 1995) and references therein). We measure the local fitness as a combination of the local error and size of the subtree, similar to the global fitness (defined as Eqn. 22). The use of the size term biases evolution to choose smaller building blocks against complex ones.

From two parent trees A_i and B_i, the offspring C_i is generated by the following procedure (see Figure 4):

1. Select a subtree a of A_i the one that has worst (largest) local fitness value.
2. Select a subtree b of B_i the one that has best (smallest) local fitness value.
3. Produce an offspring C_i by copying A_i and replacing a with b.
4. Consider all subtrees whose root lies on the path from b to the root of C_i as potential building blocks.

The first two steps involve choosing crossover points. A similar method for guiding subtree crossover was also employed by Iba (Iba and de Garis, 1996). The last step is used in conjunction with the library of building blocks.

Building blocks in the population are discovered and reused during a run. To avoid the combinatorial explosion of the candidates we confine them as the offspring C_i's subtrees whose root lies on the path from b to the root of C_i. If the local fitness of the subtree exceeds a threshold, then it is added to the library. The size of the library is limited. If the library is full and a candidate is found fitter than the worst in the library, then the candidate replaces the worst element. This allows "forgetting" of less fit substructures for the sake of better fit building blocks. Similar concepts has

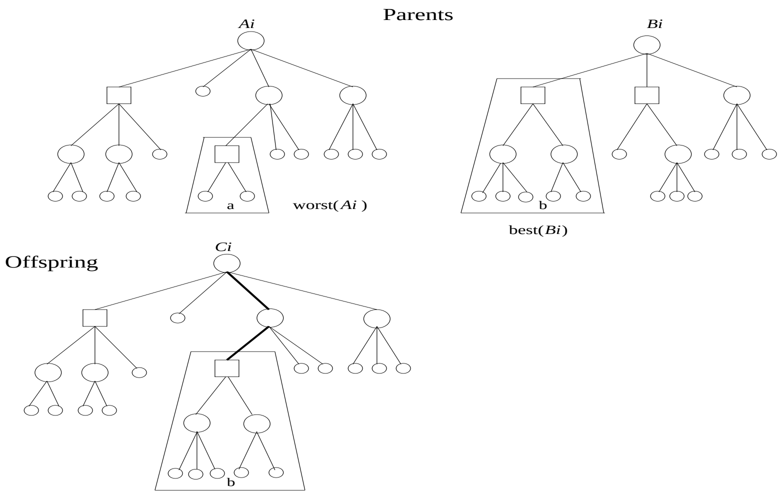


Figure 4: *Fitness based crossover and the search for building blocks. (bottom) The offspring C_i's subtrees whose root lies on the path from b to the root of C_i are tested for being added to the library, as marked by thick lines.*

been employed in GLiB (Angeline and Pollack, 1993), ARL (Rosca, 1995), and ADFs (Koza, 1993; O'Reilly, 1996) to promote the induction of modular solutions.

The library elements are used in a similar way that crossover is performed: an element a of the library is substituted for a single node of randomly chosen individual in the population. A possible side-effect of this building-block injection operation is a tendency to increase tree size; since there are generally more nodes in lower layers (near the terminals) than in upper layers (near the root node) and the substitution of library elements for nodes in lower layers increases an individual's size, the individuals might tend to grow by building-block injection. As a counter-operation to the growing tendency, a heuristic for explicitly pruning the trees is also implemented. When pruning, a subtree is replaced by its descendent subtree if the descendent subtree is fitter than the subtree itself.

Three different types of mutations are distinguished. First, mutation is used to change the neuron type. This is performed by randomly choosing a neuron type from the function set $\mathcal{F}$ which is different from the current neuron type. For instance, a sigma unit can be changed to a pi unit and vice versa. The second type of mutation is to change the input unit label. Here, an index value is randomly chosen from the terminal set $\mathcal{T}$ which is different from the current index value assigned to the input node. The third kind of mutation is used to modify weight values. This is applied in conjunction with local search for weight adaptation as described in the next subsection.

4.3 BGA-Based Parametric Learning

Adaptation of weights and biases is performed by local search. Because of the generality of neuron types, we use for local search another evolutionary method that does not make any limiting assumptions, such as continuity or differentiability, required for neural net training algorithms. During the local search, the structure of the network is fixed. The search attempts to find only a rough approximation of local optima since a perfect search would be too expensive, considering the fact that the network undergoes a structural change in the next generation.

A local search for a network consists of a number of LS_{max} iterations at most. Each iteration applies only parametric mutation that perturbs the weight vector $\mathbf{w}$ of A_i with exponential noise, a method used by BGA (Mühlenbein and Schlierkamp-Voosen, 1994). If the newly generated network A_i' is fitter than the old one A_i, then the new one is set as the current network and the next iteration continues. The local search can terminate before LS_{max} iterations if no better weight vector is found for a significantly long time. Typically, LS_{max} was set 10 to 100 in experiments.

A new weight vector $\mathbf{w}$ is generated by applying a gene mutation to each element w_i with the probability of μ_{weight}. The mutation of a gene is performed by adding a value from the interval $[-R, R]$, where the range R denotes the maximum size of mutation steps. The new weight value w_i' of w_i is computed as follows:

$$w_i' = w_i \pm R_i \cdot \delta, \quad \text{with } \delta = 2^{-K \cdot \eta} \text{ and } \eta \in [0, 1]. \tag{14}$$

The random number η is chosen from a uniform distribution over $[0, 1]$. The density function $\phi(Z)$ and the distribution function $\Phi(Z)$ for the random variable $Z = 2^{-K \cdot \eta}$ are given as:

$$\phi(z) = \frac{1}{K \cdot \ln(2) \cdot z} \tag{15}$$

$$\Phi(z) = Pr(Z \leq z) = \int_{2^{-K}}^{z} \phi(u)du = 1 + \frac{1}{K \cdot \ln(2)} \cdot \ln(z) \tag{16}$$

The exponential characteristic of $\delta = 2^{-K \cdot \eta} \in [2^{-K}, 1]$ ensures that the mutation step size of maximum vale R_i is possible, but small steps are more frequent. The constant K determines the shape of the exponentional function and thus influences the probability of choosing large mutation steps: the larger the value K, the less the probability of taking large steps. K also determines the smallest step size $R_i \cdot 2^{-K}$. The exponential mutation is contrasted with the Gaussian mutation implemented in evolution strategies and evolutionary programming (Fogel, Fogel and Porto, 1990) such that very large steps take place with only a small probability.

Due to the large costs for local search, we have used various heuristics for applying local search. One heuristic is to use local search immediately after fitness evaluation to some portion, say top 50%, of the population instead of all its members. Another heuristic is to adapt the intensity of local search, LS_{max}, during a run. For instance, a small number of local search steps is used during the earlier phase, while a large number of local search steps are used after fit individuals have appeared. A combination of both strategies is also possible. For instance, local search is applied with low intensity at each generation only to 50% of the whole population until a specified fitness is reached, after which local search alone is intensively applied to the best 10% of the population.

4.4 Occam's Razor – MDL-Based Fitness Function

Our objective is to find a neural program or model A whose evaluation $f_A(\mathbf{x})$ best approximates the desired relation $\tilde{f}(\mathbf{x})$ given an input $\mathbf{x}$. The data misfit of the program for the dataset D is measured by $E(D|A) = \frac{1}{N}\sum_{c=1}^{N}\left(\mathbf{y}_c - f_A(\mathbf{x}_c)\right)^2$, where N is the number of training examples. Considering the program as a Gaussian model of the data, the likelihood of the training data is described by $P(D|A) = \frac{1}{Z(\beta)}\exp(-\beta E(D|A))$, where $Z(\beta)$ is a normalizing constant, and β is a positive constant determining the sensitivity of the probability to the error value.

The Bayesian model-comparison techniques choose between alternative models by trading off the simplicity of a model against the associated data misfit. Thus it is reasonable to define genetic programming as the maximization of the posterior probability:

$$A_{best} = \arg\max_{A_i \in \mathcal{A}} \{P(A_i|D)\} = \arg\max_{A_i \in \mathcal{A}} \{P(D|A_i)P(A_i)\} \tag{17}$$

where arg function takes its argument as its result. Here the Bayes' rule $P(A|D) = P(D|A)P(A)/P(D)$ is used.

Though the Bayesian inference is very useful in theory, it is not very convenient to deal with in practice. Alternatively, we can use the model complexity; according to coding theory (Rissanen, 1986), if $P(\mathbf{x})$ is given, then its code length is given as $L(P(\mathbf{x})) = -\log(P(\mathbf{x}))$. Maximizing $P(D|A)P(A)$ is thus equivalent to minimizing code length:

$$L(A|D) \;=\; L(P(D|A)P(A)) = -\log(P(D|A)P(A)) = L(D|A) + L(A), \tag{18}$$

where $L(D|A) = -\log P(D|A)$ and $L(A) = -\log P(A)$. Here $L(D|A)$ is the code length of the data when encoded using the model A as a predictor for the data D, and $L(A)$ is the length of the model itself. This leads to the minimum description length (MDL) principle (Rissanen, 1986). The idea is to estimate the simplest density that has high likelihood by minimizing the total length of the description of the data:

$$A_{best} = \arg \min_{A_i \in \mathcal{A}} \{L(A_i|D)\} = \arg \min_{A_i \in \mathcal{A}} \{L(D|A_i) + L(A_i)\}. \tag{19}$$

An implementation of MDL typically requires knowing the true underlying probability distribution or an approximation of it. In general, however, the distribution of underlying data structure is unknown and the exact formula for the fitness function is impossible to obtain. This is why "pure" MDL approaches have some difficulty and should be used with care in genetic programming.

Statistical theories suggest that too small a program lacks the learning capability while too large a program may generalize poorly on unseen data (Geman, Bienenstock and Doursat, 1992). A finite set of search points and the maximum depth of trees are usually set as user-defined parameters in order to control tree sizes, but an appropriate depth is not known beforehand. What we need in practice is a general mechanism that can flexibly control the program complexity to find the most parsimonious program while satisfying the desirable training accuracy.

For an effective balancing of accuracy and parsimony of evolved structures, we proposed in (Zhang and Mühlenbein, 1995) to measure the fitness of a program A given a training set D in its most general form as

$$F(A|D) = F_D + F_A = \beta E(D|A) + \alpha C(A), \tag{20}$$

where the parameters α and β control the trade-off between complexity $C(A)$ and fitting error $E(D|A)$ of the program. In this framework, genetic programming is considered as a search for a program that minimizes $F(A|D)$, or

$$A_{best} = \arg \min_{A_i \in \mathcal{A}} \{F(A_i|D)\} = \arg \min_{A_i \in \mathcal{A}} \{\beta E(D|A_i) + \alpha C(A_i)\}. \tag{21}$$

Now we describe a general adaptive technique that balances α and β in unknown environments. Care must be taken in applying the MDL principle to genetic programming so that redundant structures are pruned when possible, but at the same time premature convergence is avoided. Avoiding the loss of diversity is especially important in early generations, while strong pruning is desirable to get parsimonious solutions and improve generalization performance later in the run.

To balance the parsimony with accuracy dynamically, we fix the error factor β at each generation and change the complexity factor α adaptively. Let $E_i(g)$ and $C_i(g)$ denote the error and complexity of ith individual at generation g. The raw complexity of neural trees can be measured as the sum of the number of units, weights, and layers. The fitness of an individual i at generation g is defined as follows:

$$F_i(g) \quad = \quad E_i(g) + \alpha(g) C_i(g), \tag{22}$$

where $0 \leq E_i(g) \leq 1$ and $C_i(g) > 0$ are assumed. Here $\alpha(g)$ is called the adaptive Occam factor and expressed as

$$\alpha(g) = \begin{cases} \frac{1}{N^2} \frac{E_{best}(g-1)}{\hat{C}_{best}(g)} & \text{if } E_{best}(g-1) > \epsilon \\ \frac{1}{N^2} \frac{1}{E_{best}(g-1) \cdot \hat{C}_{best}(g)} & \text{otherwise,} \end{cases} \tag{23}$$

where N is the size of training set. User-defined constant ϵ specifies the maximum training error allowed for the final solution.

Note that $\alpha(g)$ depends on $E_{best}(g-1)$ and $\hat{C}_{best}(g)$. $E_{best}(g-1)$ is the error of the best program of generation $g-1$. $\hat{C}_{best}(g)$ is the size of the best program at generation g estimated at generation $g-1$ (see (Zhang and Mühlenbein, 1995) for more details). $\hat{C}_{best}(g)$ is used for the normalization of the complexity factor. In essence, two adaptation phases are distinguished. When $E_{best}(g-1) > \epsilon$, $\alpha(g)$ decreases as the training error falls since $E_{best}(g-1) \leq 1$. This encourages fast error reduction at the early stages of evolution. For $E_{best}(g-1) \leq \epsilon$, in contrast, as $E_{best}(g)$ approaches 0 the relative importance of complexity increases due to $E_{best}(g-1) \ll 1$. This emphasizes stronger complexity reduction at the final stages to obtain parsimonious solutions.

Iba et al. (Iba, Kurita, de Garis and Sato, 1993) have used a pure MDL criterion for genetic programming. They defined fitness as simply the sum of error and complexity costs, followed by a normalization of the total costs. Therefore, the complexity value is as important as the error value in determining the total fitness value of an individual. This works perfectly when the coding scheme exactly reflects the true probability distribution of the environment. One drawback in this implementation of the MDL principle is the lack of flexibility in balancing accuracy with parsimony in unknown environments. That is, there is a risk that the tree size may be penalized too much, resulting in premature convergence. In fact, the authors remark that this kind of MDL approach should be used carefully when evolving other structures. In contrast, the adaptive Occam method is a general adaptive technique that balances parsimony and complexity of programs in an unknown environment.

5 Experimental Results and Analysis

The goal in this section is to demonstrate the performance of the proposed method to solve problems of practical value. The first two subsections report the experimental results in comparison to previous work. The last two subsections analyze and discuss the properties of the method.

5.1 Mackey-Glass Time Series

One often used benchmark problems for time-series prediction is the Mackey-Glass differential equation, a dynamical system created as a model for blood flow:

$$\frac{dx(t)}{dt} = \frac{ax(t-\tau)}{1+x^{10}(t-\tau)} - bx(t). \tag{24}$$

For $a = 0.2$, $b = 0.1$, and $\tau = 17$, the trajectory is chaotic and lies on an attractor of fractal dimension of approximately 2.1. A time series from this system is plotted in Figure 5. Iba et al. (1993) uses this problem for the demonstration of STROGANOFF (Iba, Kurita, de Garis and Sato, 1993). Mulloy et al. (1996) solves this problem by genetic programming based on a function set consisting of $\{+, -, *, \%, sin, cos, k^{10}\}$. Previously, Oakley (1994) experimented with an instance of this problem with $\tau = 30$ using a function set of $\{+, -, *, /, sin, cos, exp10\}$.

Since the initial values $x(0)$ to $x(17)$ were not given in the literature and obviously different authors have used different initial values, we have chosen the values so that the time series looks most like that of the method in comparison: $x_0 = 1.000$, $x_1 = 1.002$, $x_2 = 1.000$, $x_3 = 0.992$, $x_4 = 0.983$, $x_5 = 0.973$, $x_6 = 0.966$, $x_7 = 0.963$, $x_8 = 0.972$, $x_9 = 0.987$, $x_{10} = 1.004$, $x_{11} = 1.025$, $x_{12} = 1.049$, $x_{13} = 1.077$, $x_{14} = 1.101$, $x_{15} = 1.123$, $x_{16} = 1.139$, and $x_{17} = 1.133$.

Figure 6 (left) shows the learning accuracy of the training data, $x(t)$ values divided by 10, for the one-step ahead prediction. Here the problem is to learn

$$x(t+10) = \varphi((x(t), x(t+1), ..., x(t+9)). \tag{25}$$

STROGANOFF solved this problem with a prediction error of 4.7×10^{-6} (Iba, Kurita, de Garis and Sato, 1993). It used second order polynomials and took 1740 generations with population size of 60. The solution was 13 polynomials of degree 2, containing 6 parameters each and 78 parameters in total. Parameter fitting of the polynomials was performed by the least mean squares method.

Figure 6 (right) plots our results on the one-step ahead prediction for the test data which is the continuing series. This run was made with population size of $M = 200$ for 80 generations.

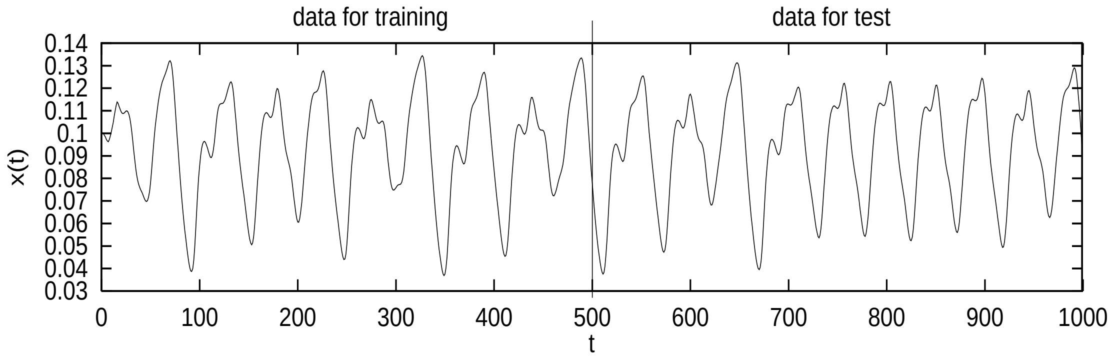


Figure 5: *Mackey-Glass time series.*

Method	Pop size	Generation	Prediction error	Units	Hiddens	Layers
STROGANOFF	80	1740	4.7×10^{-6}	78	13	8
Neural Trees	200	60	0.6×10^{-6}	59	13	4

Table 1: *Comparison of prediction results for the test data of the Mackey-Glass problem.*

The neural tree structure evolved for the above performance contained 59 weights with 13 hidden units. The mean squared error of this solution is 0.6×10^{-6}. This error is eight times lower than the result of STROGANOFF described above. The results are summarized in Table 1.

Figure 7 (top) shows the neural tree evolved to generate the series in Figure 6 in comparison with the structure evolved by STROGANOFF, Figure 7(bottom right). Figure 7 (bottom left) depicts another neural tree which results in a bit worse performance but has a very sparse structure. The evolutionary method seems to assign different credit to different inputs to find problem-specific solutions. It is interesting to observe in the neural tree solution that the inputs of the near past, such as $x(10), x(9), x(8)$ etc., appear more often than the inputs with greater distance in the series. And in general $x(t-1)$ appeared more often than the other inputs in solutions evolved in most experiments.

This property of automatic architecture (feature) selection is an advantage of genetic programming over the standard backpropagation in which an appropriate structure must be provided by the designer. In another set of experiments, we found that, though backpropagation networks could achieve predictive accuracy comparable to the best neural tree, they required a good network

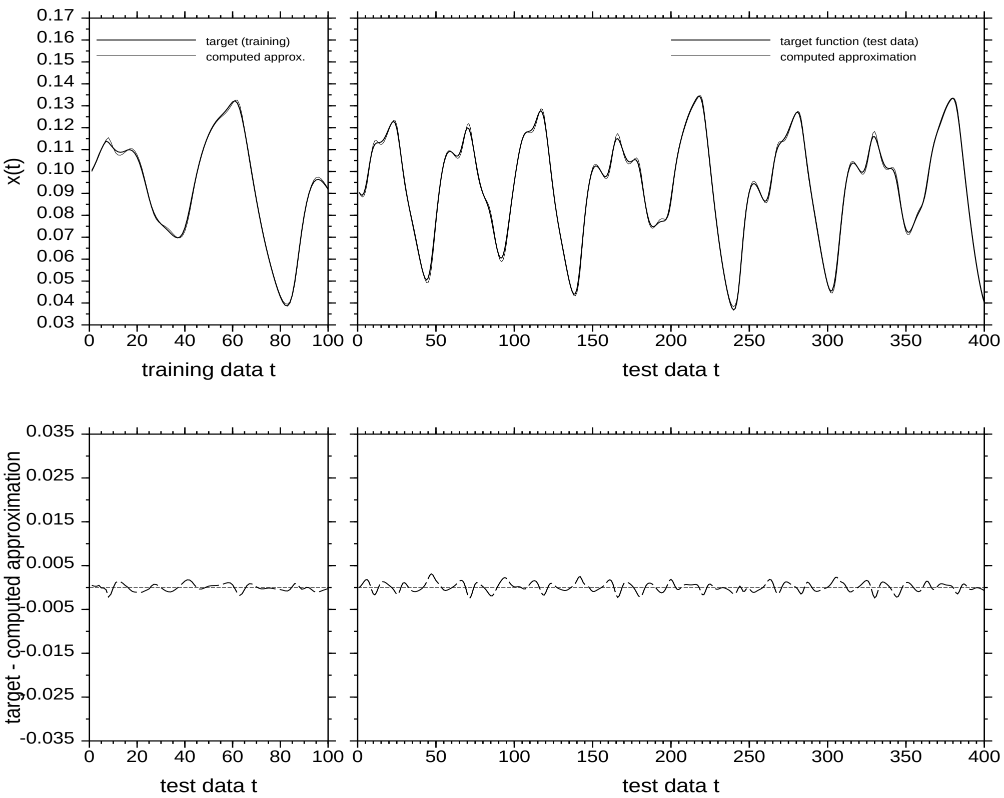


Figure 6: *Results for the Mackey-Glass time series data: (top left) training performance, (top right) one-step ahead prediction performance. The thick curves denote the true values and the thin curves the predicted. (bottom) The plots show the difference between the true and predicted values. The maximum difference is* ± 0.003.

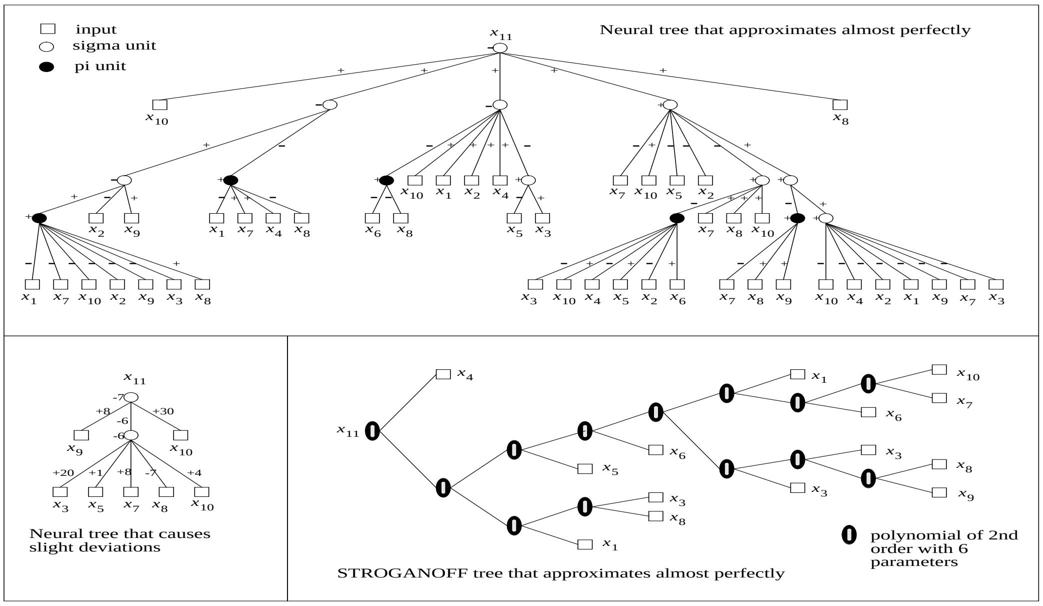


Figure 7: *Sample solutions for the Mackey-Glass time series: (top and bottom left) neural tree solutions, (bottom right) the solution obtained by STROGANOFF.*

structure which usually contained more parameters than the best neural trees. Detailed results are shown in Table 2. Here the performance was measured in terms of the root mean square error divided by the standard deviation of the data.

5.2 Chaos in a Far-Infrared NH_3 Laser

Figure 8 shows a series of 1000 measurements of chaotic intensity fluctuations. This data set was generated by sampling every other data points from far-infrared NH_3 laser data in a physics laboratory. The prediction of the data is difficult enough to have been used as a benchmark problem in the 1992 Santa Fe time-series competition (Hübner and Tang, 1993). Though the time series can approximately be described by three coupled nonlinear ordinary differential equations, the first 500 data points used for training show only two of five collapses, and thus the prediction of the next collapses is a difficult task based on so few instances. Though other methods, such as state-space reconstruction and standard backpropagation networks, could achieve satisfactory results for this problem, their design required careful analysis of the problem. The main objective of our experiments is to demonstrate a comparable performance of the neural trees designed automatically

Method	Hidden units	Num. weights	Training error	Prediction error
Neural Trees	30	153	0.52	0.58
Backpropagation 1	100	601	0.53	0.56
Backpropagation 2	300	1801	0.69	0.84

Table 2: *Comparison of neural tree results against backpropagation networks for another set of Mackey-Glass experiments. Neural trees were evolved for 80 generations with population size of 100.*

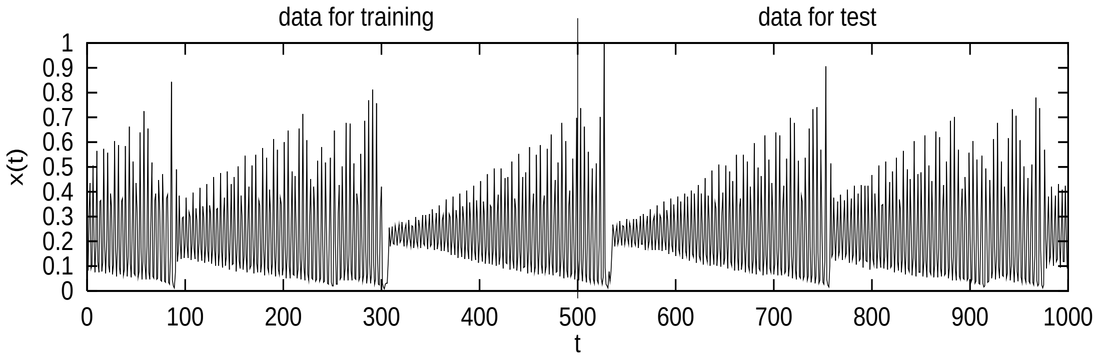


Figure 8: *Laser intensity time series.*

by an evolutionary process.

The training set was generated from the time series as follows: three contiguous values $x_1(t)$, $x_2(t)$, $x_3(t)$ were used as input for the tth training pattern and the immediate next point $x_4(t)$ was used as the target value $y(t)$ to be predicted. Figure 9(top) shows the learning accuracy for the training data which contained 500 data points. Figure 9(middle) plots the one-step ahead prediction for the next 500 test points which were not shown during the training phase. Figure 9(bottom) is the difference between the true output and the predicted output for the test data. At the collapse points near $t = 30, 260, 470$, larger differences are observed than at the other data points. This is because the training data contained only two collapse points. This run was made with a population size of $M = 250$ for 150 generations. The parameter LS_{max} was set to 20.

The convergence characteristics of the best and worst runs out of 20 runs for learning the laser time series are depicted in Figure 10. The left figure depicts the evolution of fitness and error,

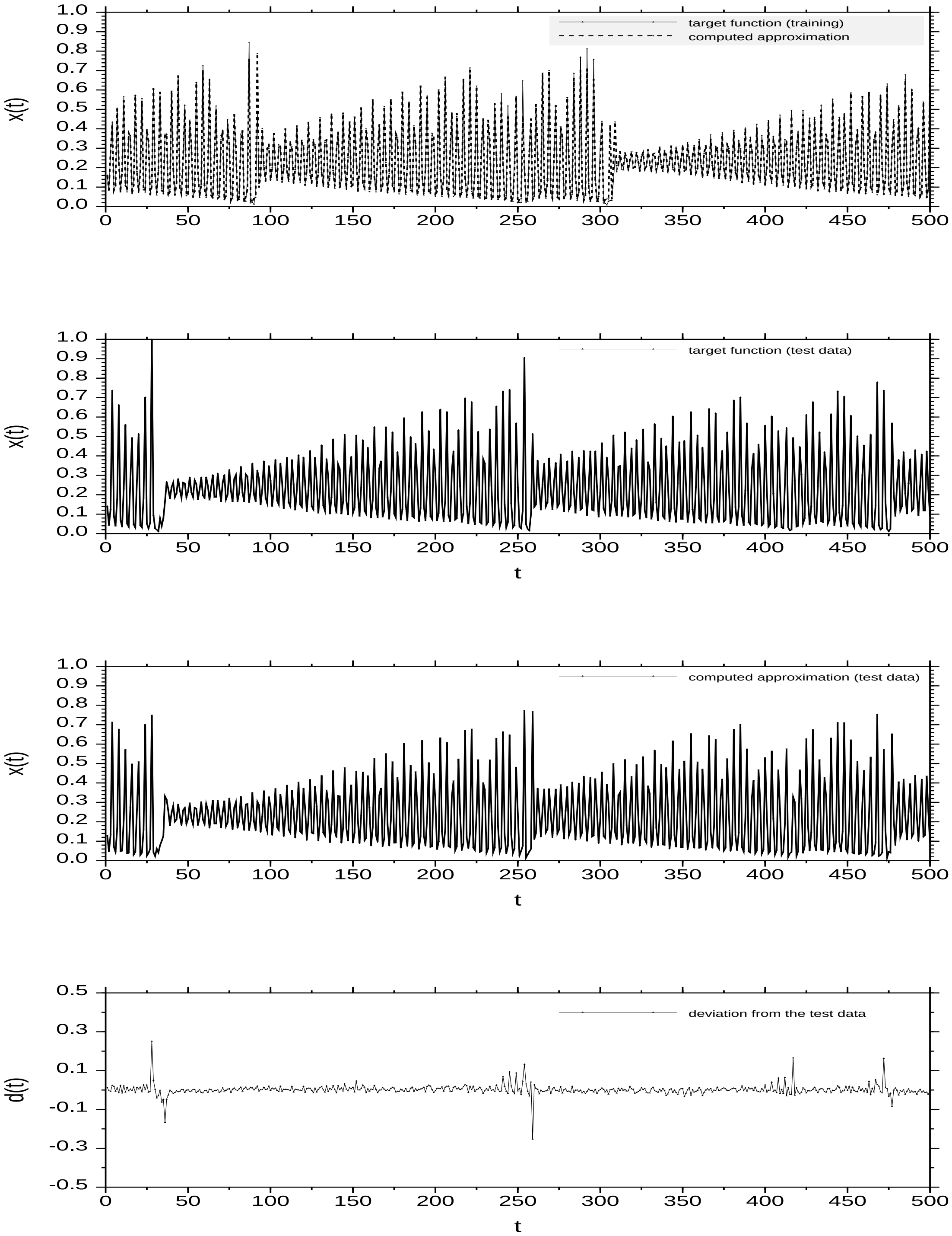


Figure 9: *Results for the laser intensity time series: (top) performance for the training data, (middle) prediction performance for the test data, (bottom) difference between true values and predicted values for the test data.*

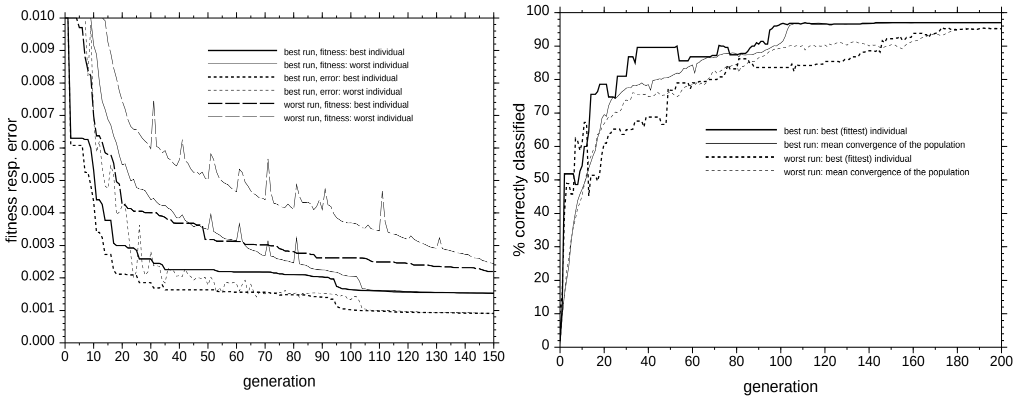


Figure 10: *The evolution of performance for the best and worst runs for the laser data: (left) fitness and error for the best and worst individuals, (right) performance in percent of correct classifications.*

whereas the right figure shows the results in terms of correct classification rate. The prediction output was counted as correct when it falls within the acceptance interval of the true value. Due to the elitist selection strategy, the fitness values of the best individuals decrease monotonically.

It can be observed in Figure 10 (left) that the difference between fitness and error is large in early generations while it gets smaller as generation goes on. This indicates that the early stage focuses more on structural adaptation rather than parametric learning, the role of which reverses in the later stage. This is a reason why, given a limited resource, weight mutations should be intensively used instead of structural crossovers. The fast decrease in fitness values around generation 100 in Figure 10 (left) is due to the increased intensity of local search, $LS_{max} = 1000$, after reaching the fitness threshold. The corresponding improvement in classification rate is also seen in the right figure.

5.3 Occam's Razor and Solution Quality

Occam's Razor is a principle of parsimony that says the simpler model should be preferred to complex models if all other things are equal. The effect of the Occam factor was investigated on the solution accuracy and complexity. In particular, we analyzed the generalization performance of various complex trees for the prediction of the laser and Mackey-Glass time series. A more broad and detailed treatment of this issue can be found in our previous work (Zhang and Mühlenbein,

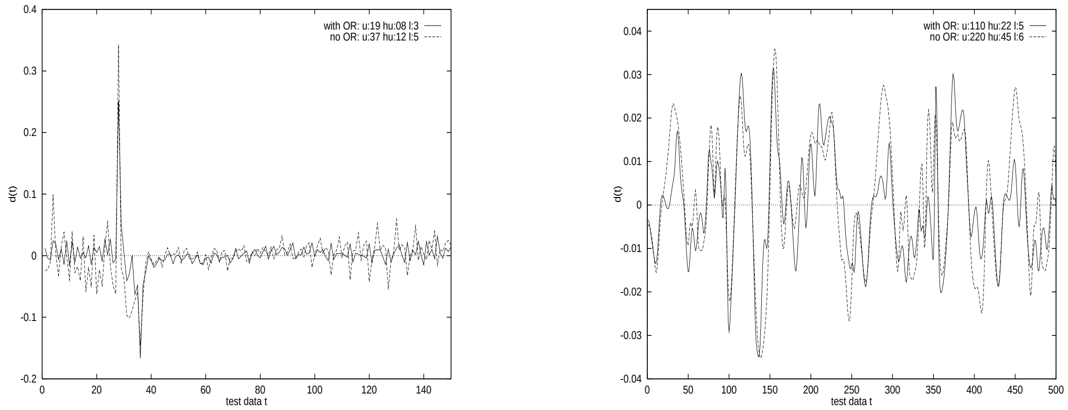


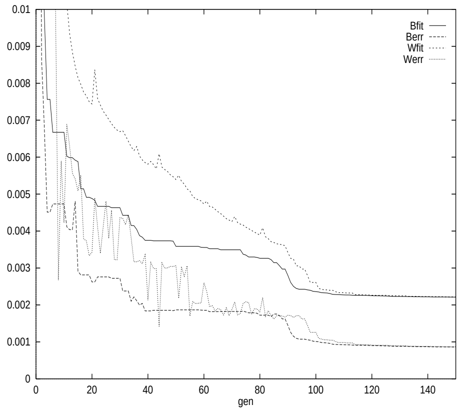
Figure 11: *Generalization performance of two solutions with differing complexity: (left) laser data, (right) prediction of Mackey-Glass series $x(t)$ from $x(t-85)$, $x(t-91)$, and $x(t-97)$.*

1995).

Figure 11 (left) compares the generalization accuracy for two different runs with the laser data. One run was made using Occam's Razor in its fitness function (MDL-based fitness measure) and resulted in a relatively small neural tree structure of 3 layers containing 8 hidden units and 19 units in total. The other run was made without using complexity penalty (but used also a maximum complexity limit), resulting in a larger structure of 5 layers containing 12 hidden units and total 37 units. The use of complexity penalty evolved a more parsimonious structure. The dotted line in Figure 11 (left) shows the error made by the large tree, which is generally larger than the error made by the smaller tree.

Figure 11 (right) plots similar results obtained for the Mackey-Glass time series data. Here the problem was to predict $x(t)$ from $x(t-85)$, $x(t-91)$, and $x(t-97)$ and the tree with 220 units caused larger generalization error than the tree with 110 units. The results demonstrate that the generalization performance of smaller trees are better than the larger ones and the use of Occam's Razor is useful for improving generalization accuracy. The generality of this argument both in theory and practice was extensively studied in (Zhang and Mühlenbein, 1995).

Figure 12 (left) plots the evolution of fitness and error of the best and worst individuals in each generation. The fast reduction in the error curves after generation 90 is due to the intensive application of local search. The evolution of program complexity in terms of the number of layers was shown in Figure 12 (right). Comparing both figures, the tendency is observed that the program



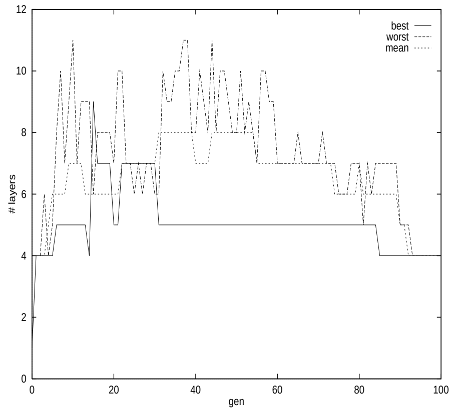


Figure 12: *Evolution of fitness and complexity for the laser data. For this experiment, the local search was applied after generation 90, before which crossover and mutation were used as well. (left) The top two curves plot the fitness of the worst and best individuals, and the bottom two curves show the error of the worst and best individuals, respectively. (right) The top and bottom curves show the size of worst and best individuals while the middle curve is the mean size of the population.*

size first grows and then shrinks during which the error of the best individual steadily decreases. This demonstrates the desired effect of the Occam's Razor implemented in our MDL-based adaptive fitness function; it promotes the trees to grow when significant error reduction is required, while it prefers smaller trees to larger ones when their errors are comparable.

5.4 Library of Building Blocks

The library can be considered as a population of partial solutions that gets fitter during evolution. New fitter solutions are stored and old worse subsolutions get replaced. The library plays the role of a kind of memory for good partial solutions and complements the genetic material in the population.

Figure 13 compares the final results with and without the library for the infrared laser problem. The comparison was made in terms of best and population-average performances for 40 runs each. A clear improvement of performance was achieved by using the library both for the best individual and the average performance of the population.

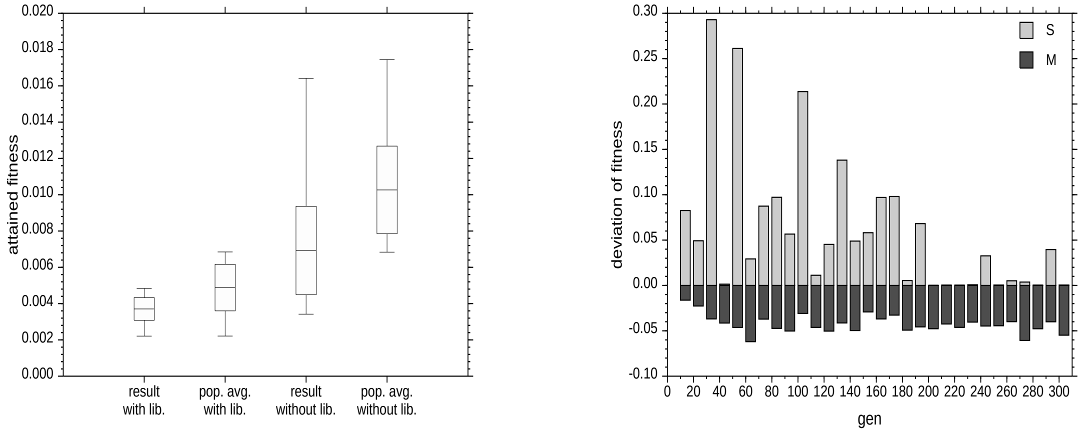


Figure 13: *The effect of library. (left) The leftmost two bars show the best and average fitness with library while the rightmost two bars are the best and average fitness without library. 40 runs were made for each. (right) 'S' bars indicate the best positive effect on fitness difference immediately after injection of building blocks, and 'M' bars denote the negative effect on the difference in average fitness.*

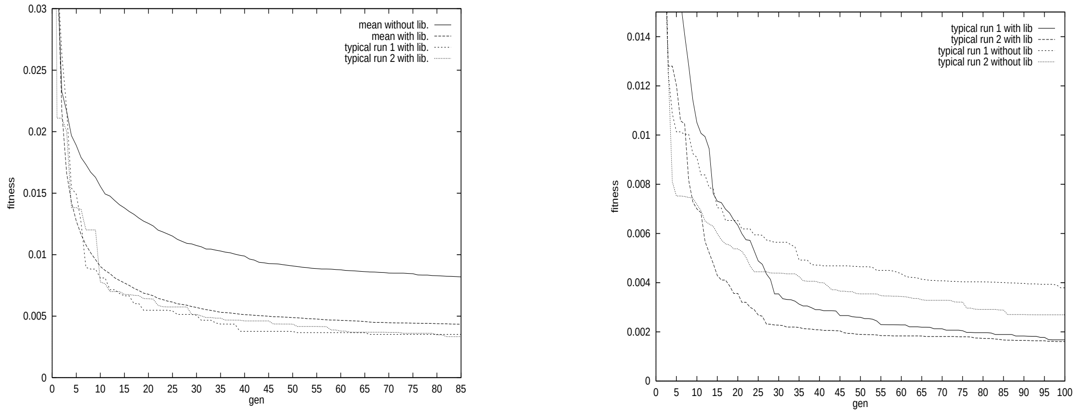


Figure 14: *Comparison of fitness evolution with and without library. (left) The top two curves plot the average fitness of best individuals for 50 runs on the laser data: the upper one without using library, the lower one with library. The bottom two curves are results for typical runs. (right) The top two curves show the best fitness evolution without library and the bottom two the results of typical runs with library for learning a noisy polynomial function of degree three.*

For a more detailed analysis of the effect of the library, we examined the improvement of fitness values "immediately" after the insertion of building blocks. We also measured the average improvement of all individuals every ten generations. It turned out that the best performance was significantly improved but the average fitness was decreased. The negative effect on the average performance seems attributed to the fact that the performance was measured immediately after the building block insertion. The longer term effects, however, tend to be positive since the weights of inserted subtree get fitted to the new context during further generations.

Figure 14 plots the fitness evolution for runs with library in comparison to those without using library. The top two curves of the left figure compare the average fitness of best individuals for 50 runs on the laser data, showing a clear positive effect on convergence speed of using library. Also shown are two typical runs with library. In many other applications we examined, the use of library has lead to a better solution in terms of both accuracy and parsimony, as exemplified in Figure 14 (right). Here the top two curves plot the best fitness without library while the bottom two show the results with library for learning a noisy polynomial function of degree three.

6 Conclusions

A hybrid evolutionary framework was presented which combines genetic programming and the breeder genetic algorithm for the induction of a broad class of higher-order neural networks. The neural tree encoding scheme was introduced and successfully applied to time-series prediction problems of practical interest.

Though the presented method can best be described in the genetic programming paradigm, it has several interesting features that extend the standard genetic programming method. As opposed to other GP-evolved programs, the neural programs contain a large number of numerical variables as well as symbolic, posing a situation where structures and parameters have to co-evolve. The present work exemplifies how to effectively extend the dynamic structure adaptation capability of GP through another evolutionary algorithm for parametric learning.

Empirical studies have shown that the early stage focuses more on structural adaptation rather than parametric learning, the role of which reverses later in the run. Considering this observation and the fact that local search is expensive, we have applied the local search to some portion of the population, instead of all its members. The intensity of local search has also been adapted during the run. Generally, it is recommended to use a small LS_{max} to a large portion of the population early in the run and to use a large LS_{max} for a small portion of the population later in the run.

The generality of our approach is contrasted with various existing methods for the synthesis of higher-order networks which investigate only restricted topological subsets rather than the complete class of network architectures. This rather ambitious goal (see Appendix A for a complexity analysis of the problem) could be achieved by taking an evolutionary approach which allows minimal representational constraints. The problem of "bloating" or unlimited growth of programs was dealt with by MDL-based parsimony control. The efficiency of the tree induction process was further enhanced by incorporating several mechanisms, such as libraries of building blocks, local fitness-dependent crossover, injection and pruning of building blocks as well as scheduling of the intensity and sequence of genetic operators.

Acknowledgments

Thanks to Peter J. Angeline and anonymous reviewers for invaluable comments on previous versions of this paper. This research was supported in part by grants from Real World Computing Partnership (RWCP) and KOSEF (961-0901-001-2).

References

Angeline, P. J. and Pollack, J. B. (1993). Coevolving high-level representations. In *Artificial Life III*. Cambridge, MA: MIT Press.

Angeline, P. J., Saunders, G. M. and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Trans. on Neural Networks*, 5(1):54–65.

Balakrishnan, K. and Honavar, V. (1995). Evolutionary design of neural architectures. Technical report, CS-TR-95-01, AI Lab, Dept. of Computer Science, Iowa State University.

Belew, R. K., McInerney, J. and Schraudolph, N. N. (1992). Evolving networks: using genetic algorithms with connectionist learning.

Fahner, G. and Eckmiller, R. (1994). Structural adaptation of parsimonious higher-order neural classifiers. *Neural Networks*, 7(2):279–289.

Fogel, D. B., Fogel, L. J. and Porto, V. W. (1990). Evolving neural networks. *Biological Cybernetics*, 63:487–493.

Geman, S., Bienenstock, E. and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58.

Giles, C. L. and Maxwell, T. (1987). Learning, invariance, and generalization in high-order neural networks. *Applied Optics*, 26(23):4972–4978.

Goldberg, D. E. (1989). Genetic algorithms and walsh functions. *Complex Systems*, 3:129–152.

Gruau, F. (1993). Genetic synthesis of modular neural networks. In *Proc. Fifth Int. Conf. Genetic Algorithms*, pages 318–325. Morgan Kaufmann.

Hübner, H., Weiss, C. O., Abraham, N. B. and Tang, D. (1993). Lorenz-like chaos in nh_3-fir laser. In Weigend, A. and Gershenfeld, N., editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 73–104. Addison-Wesley.

Iba, H., Kurita, T., de Garis, H. and Sato, T. (1993). System identification using structured genetic algorithms. In *Proc. Fifth Int. Conf. Genetic Algorithms*, pages 279–286. Morgan Kaufmann.

Iba, H. and de Garis, H. (1996). Extending genetic programming with recombinative guidance. In Angeline, P. J. and Kinnear, K. E., editors, *Advances in Genetic Programming 2*, pages 69–88. Cambridge, MA: MIT Press.

Ivanknenko, A. G. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man and Cybernetics*, 1(4):364–378.

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

Kowalczyk, A. and Ferra, H. L. (1994). Developing higher-order networks with empirically selected units. *IEEE Trans. on Neural Networks*, 5(5):698–711.

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

Koza, J. R. (1993). Simultaneous discovery of reusable detectors and subroutines using genetic programming. In *Proc. Fifth Int. Conf. on Genetic Algorithms*, pages 295–302. Morgan Kaufmann.

Minsky, M. L. and Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Mühlenbein, H. and Schlierkamp-Voosen, D. (1994). The science of breeding and its application to the breeder genetic algorithm. *Evolutionary Computation*, 1(4):335–360.

O'Reilly, U.-M. (1996). Investigating the generality of automatically defined functions. In D. E. Goldberg, D. B. F. and Riolo, E. L., editors, *Proc. First Annual Conf. on Genetic Programming*, pages 351–356. Cambridge, MA: MIT Press,.

Rissanen, J. (1986). Stochastic complexity and modeling. *The Annals of Statistics*, 14:1080–1100.

Rosca, J. (1995). Towards automatic discovery of building blocks in genetic programming. In Rosca, J., editor, *Proc. 1995 AAAI Fall Symposium on Genetic Programming*, pages 78–85. Menlo Park, CA: AAAI Press.

Zhang, B. T. and Muehlenbein, H. (1994). Synthesis of sigma-pi neural networks by the breeder genetic programming. In *Proc. IEEE Int. Conf. on Evolutionary Computation*, pages 318–324. New York, NY: IEEE Computer Society Press.

Zhang, B. T. and Mühlenbein, H. (1995). Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38.

Zhang, B. T. and Mühlenbein, H. (1996). Adaptive fitness functions for dynamic growing/pruning of program trees. In Angeline, P. J. and Kinnear, K. E., editors, *Advances in Genetic Programming 2*, pages 241–256. Cambridge, MA: MIT Press.

Appendix A: Theoretical Complexity of Neural Tree Induction

We analyze computational requirements for solving the problem of neural tree induction. For simplicity, our analysis focuses on the set $\mathcal{NT}(d,b)$ of complete neural trees of depth d and braching factor b. The size of architecture space will first be calculated and then the size of parameter space.

Consider a complete tree of depth d and branching factor b. The number of nonterminal nodes in this tree structure is

$$1 + b^1 + b^2 + b^3 + \cdots + b^{d-1} = \frac{b^d - 1}{b - 1}. \tag{26}$$

The number of terminal nodes is b^d. An instance of the tree consists of the nonterminal nodes with associated labels chosen from the function set $\mathcal{F}$ and the terminal nodes instantiated with labels from the terminal set $\mathcal{T}$. Thus, the number of possible architectures $\mathcal{NT}(d,b)$ is given by

$$|\mathcal{A}| = |\mathcal{F}|^{\frac{b^d-1}{b-1}} \cdot |\mathcal{T}|^{b^d} \tag{27}$$

where $|\mathcal{F}|$ and $|\mathcal{T}|$ are the sizes of $\mathcal{F}$ and $\mathcal{T}$, respectively.

The parameters in the neural trees consist of connection weights and biases. The number of weights in the complete tree of depth d and branching factor b is the same as the number of nonterminal plus terminal nodes minus one (root node):

$$b^1 + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - b}{b - 1}. \tag{28}$$

The number of biases is the same as the number of nonterminal nodes: $(b^d - 1)/(b - 1)$. Total number of parameters is the sum of both:

$$\frac{b^{d+1} - b}{b - 1} + \frac{b^d - 1}{b - 1} = \frac{b^{d+1} + b^d - b - 1}{b - 1} \tag{29}$$

If we assume for simplicity that the parameter takes a value from a set $\mathcal{V}$ of finite size $|\mathcal{V}|$, the size of the parameter space is given by

$$|\mathcal{W}| = |\mathcal{V}|^{\frac{b^{d+1}+b^d-b-1}{b-1}} \tag{30}$$

As an illustration, consider the program space of $|\mathcal{F}| = |\{\Sigma, \Pi\}| = 2$, $|\mathcal{T}| = 5$, $d = 3$, $b = 5$, and $|V| = 100$. In this case the size of structure space is $|\mathcal{A}| = 2^{31} \cdot 5^{125}$ and the size of weight space amounts to $|\mathcal{W}| = 100^{186}$, which prohibits any exhaustive search methods.