Integrated Computer-Aided Engineering 9 (2002) 73–86
IOS Press


# A Bayesian evolutionary approach to the design and learning of heterogeneous neural trees

Byoung-Tak Zhang
*Biointelligence Laboratory, School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea*
*Tel.: +82 2 880 1833; Fax: +82 2 883 3595; E-mail: btzhang@cse.snu.ac.kr*

**Abstract**: Evolutionary algorithms have been successfully applied to the design and training of neural networks, such as in optimization of network architecture, learning connection weights, and selecting training data. While most of existing evolutionary methods are focused on one of these aspects, we present in this paper an integrated approach that employs evolutionary mechanisms for the optimization of these components simultaneously. This approach is especially effective when evolving irregular, not-strictly-layered networks of heterogeneous neurons with variable receptive fields. The core of our method is the neural tree representation scheme combined with the Bayesian evolutionary learning framework. The generality and flexibility of neural trees make it easy to express and modify complex neural architectures by means of standard crossover and mutation operators. The Bayesian evolutionary framework provides a theoretical foundation for finding compact neural networks using a small data set by principled exploitation of background knowledge available in the problem domain. Performance of the presented method is demonstrated on a suite of benchmark problems and compared with those of related methods.

## 1. Introduction

Evolutionary algorithms have been successfully applied to the design and learning of neural networks. Hinton and Nowlan [14] study the interaction between learning and evolution in neural networks. Mühlenbein and Kindermann [23] suggest general schemes for evolving neural networks. Montana and Davis [22] and Fogel et al. [8] present evolutionary methods to train the connection weights of neural networks. They report some encouraging results which are comparable with conventional learning algorithms. Evolutionary algorithms have also been used to optimize the topology of neural networks that best fits to the specified task according to some explicit design criteria [32]. All these methods have been applied to conventional neural network architectures, such as strictly-layered feedforward networks.

However, some tasks can be solved much better by using network architectures which are non-conventional in terms of neuron types and network topology. An example of this class of architectures is the sigma-pi networks that have product units as well as sigma units as the primitive units [7]. Though the usefulness of these architectures are well-known, one difficulty involved with these networks is that no standard algorithms are available for their design and training [11].

In this paper, we present an evolutionary method that is suitable for the design and training of novel neural networks. The network architectures we evolve are non-conventional in the sense that neural units of heterogeneous types may be contained within a single network and the neurons are not-strictly layered and have irregular connectivity. In contrast to conventional learning algorithms, evolutionary algorithms do not require strong assumptions on the search space, such as continuity or differentiability, enabling to explore the space of novel network architectures. Our method is based on the neural tree representation scheme which is

ISSN 1069-2509/02/$8.00 

general and flexible enough to represent a broad class of network architectures and to manipulate the architectures using standard crossover and mutation operators. Feature selection problems are naturally handled by the neural trees since each node (neuron) in the tree can have a different number of children (other neurons or external inputs) and the children are modified through the usual genetic operators.

An important feature of the proposed evolutionary method is the use of an adaptive fitness function. This is based on the observation that fitness functions are essential for successful application of evolutionary algorithms to neural networks in practical applications. The enormous size of the search space prohibits straightforward evolutionary methods from successful application. Just as a good representation can transform the original, difficult problem to a more manageable problem, the choice of a good fitness function makes the search smoother.

The adaptive fitness function is based on the Bayesian inductive principle [10]. It has an error term and complexity term to effectively balance the accuracy of the network and its complexity. According to the Occam's Razor principle [36], the best neural network is the simplist one with a small number of neurons and sparse connectivity. The weights are trained not by back-propagation [28], but by another evolutionary algorithm for continuous parameter optimization. In this work we also incorporate the incremental data inheritance (IDI) method [35] into the adaptive fitness function. The IDI method starts evolution with a small data set for fitness evaluation and expands the data set as generation goes on. It proved useful for accelerating the evolution of Lisp-like symbolic programs.

The paper is organized as follows. Section 2 reviews the previous work on applying evolutionary computation to neural networks. In Section 3, we present the neural tree representation and discuss its properties. Section 4 describes the theoretical foundation of the Bayesian approach to evolving heterogeneous neural trees. Section 5 describes implementation details for evolving neural trees. Section 6 presents the experimental results. Section 7 discusses the synergy effects of evolutionary algorithms and neural networks from our experience.

## 2. Related work

Evolutionary algorithms have been used for neural networks in several applications. These include the selection of

- connection weights
- network topologies, and
- weight update rules.

Schaffer et al. [31] and Yao [33] provide comprehensive reviews of combining evolutionary algorithms and neural networks. A general way of evolving genetic neural networks was suggested by Mühlenbein and Kindermann [23]. Several early studies have used evolutionary algorithms to optimize the connection weights of neural networks [8,22]. Recent works, however, have focused on using evolutionary algorithms to optimize the network topology. Harp et al. [13] and Miller [21] 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. [32] to prune unnecessary connections. Kitano [18] and Gruau [12] have suggested encoding schemes in which a network configuration is indirectly specified by a graph generation grammar which is evolved by genetic algorithms.[1] Radi and Poli [27] use an evolutionary algorithm to discover learning rules for neural networks.

All the methods mentioned above use the back-propagation algorithm [28], a gradient-descent method, to train the weights of the network. Yao and Liu [34] present an evolutionary system that simultaneously evolves both neural network architectures and weights. Here, the architectures are modified by mutations that add or delete nodes/connections. Weights are trained by a modified back-propagation algorithm with an additive learning rate and by a simulated annealing algorithm. Koza [20] provides an alternative approach to representing neural networks, under the framework of so-called genetic programming, which enables modification not only of the weights but also of the architecture for a neural network. However, this method provides neither a general method for representing an arbitrary feedforward network, nor a mechanism for finding a network of minimal complexity. Angeline et al. [2] present an evolutionary method for constructing recurrent neural networks.

In addition to the three typical applications of evolutionary computation to neural networks, as described

[1] We use the terms evolutionary algorithms and evolutionary computation as a general concept that includes evolution strategies, evolutionary programming, genetic algorithms, genetic programming, and other computational methods gleaned from natural evolution. The term genetic algorithms will be reserved in this paper to denote the class of evolutionary algorithms that use binary strings (usually with fixed length) as their representation.

above, more recent developments also include the selection of

– neuron types
– receptive fields, and
– training data sets.

Some neuron types are more natural to represent the solution for the specific problem than others [11]. Therefore, it is natural to think of non-conventional neurons. Likewise, some input variables (features) to a neural unit are more important than others, and it is necessary to select input variables or receptive fields of units [7,19]. Selection of training data sets is also useful since the evolutionary process can be accelerated by a representative subsample of the given data set [25, 39].

One of the main issues in efficient evolution of neural networks is the representation or encoding of neural networks in genotype. Existing representation methods can be roughly divided into two categories: direct and indirect encoding [4,33]. 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 [5].

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 [18] and cellular encoding [12]. 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 [12]. However, the grammatical encoding seems only appropriate for exploring a search space having a regular structure. It seems not very suitable for a search space consisting of 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.

We present in the following section 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. Tree representations of neural networks

### *3.1. 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 $\mathcal{T} = \{x_1, x_2, \ldots, x_n\}$, where $x_i$ is the $i$th 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. 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.

Different neuron types are distinguished in the way of computing net inputs. For example, consider two different types of units: sigma and pi units. Sigma units compute the sum of weighted inputs from the lower layer:

$$\text{net}_i = \sum_j w_{ij} y_j, \tag{1}$$

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

$$\text{net}_i = \prod_j w_{ij} y_j, \tag{2}$$

where $y_j$ are the inputs to $i$. The output of a neuron is computed the sigmoid transfer function

$$y_i = f(\text{net}_i) = \frac{1}{1 + e^{-\text{net}_i}}, \tag{3}$$

where $\text{net}_i$ is the net input to the unit computed by Eqs (1) or (2).

Another type of neuron commonly used is the radial basis function (RBF) units. Given an input vector $\mathbf{x}$,

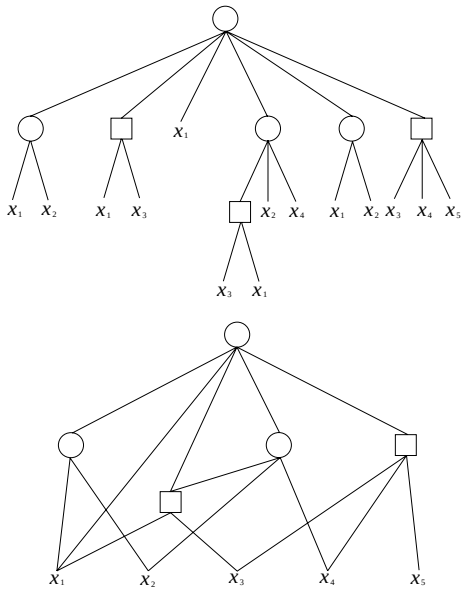


Fig. 1. Genotype (top) and phenotype (bottom) of a neural tree.

an RBF computes the output as

$$y_i = g_i(\mathbf{x}) = \exp\left\{{\sigma_i}^{-1}||\mathbf{x} - \mathbf{c}_i||\right\}, \tag{4}$$

where $\mathbf{c}_i$ and $\sigma_i$ are the center and width of the $i$th Gaussian unit. As will be clear in later sections, any other non-conventional neuron types can be employed in neural trees since they are learned by an algorithm that does not assume any strict constraints, such as continuity or differentiability, on the search space.

### *3.2. Expressive power and computational complexity*

Neural trees can represent a broad class of neural networks. In particular, it can represent complete higher-order neurons. The net input of the complete higher-order neuron is given by the multinomial expansion

$$\begin{aligned} \text{net}_i(\mathbf{x}) &= \sum_{\mathbf{a}\in\{0,1\}^n} w_{\mathbf{a}}\pi_{\mathbf{a}}(\mathbf{x}) \\ &= \sum_{\mathbf{a}\in\{0,1\}^n} w_{\mathbf{a}} \prod_{j=1}^{n} x_j^{a_j}, \end{aligned} \tag{5}$$

where $w_{\mathbf{a}} \in IR$, $\mathbf{a} = a_1 \ldots a_n \in \{0,1\}^n$, and $\mathbf{x}$ denotes the original input vector. The $2^n$ monomials $\pi_{\mathbf{a}}$ are referred to as Walsh functions. They are orthogonal and span the space of real valued functions defined over binary strings. The expansion coefficients $w_{00..00}, w_{00..01}, \ldots, w_{11..11}$ 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 [7].

Note that neural trees can represent a broad class of higher-order networks. There is no explicit limit on the order of weights. Thus, 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, pi, and any other types of units, and connections between non-neighboring layers are allowed. The network may contain partial connectivity, which is useful for the economic representation of arbitrary complex interactions.

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. Note that neural trees have a closure property, i.e. replacement of a subtree by another neural tree results in a correct neural tree structure. This is the reason why standard crossover and mutation operators can be used for adaptation of neural trees. Since subtree crossover used in genetic programming [20] applies without modification to this representation, genetic programming can be used as an evolutionary engine. However, searching the whole space of all possible neural tree is impossible in practice.

To see the complexity of the problem, we compute the size of the search space of neural trees. Consider a full tree of depth $d$ and branching factor $b$. The number of nonterminal nodes in this tree structure is

$$1 + b^1 + b^2 + b^3 + \cdots + b^{d-1} = \frac{b^d - 1}{b - 1}. \tag{6}$$

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

$$|\mathcal{A}| = |\mathcal{F}|^{\frac{b^d-1}{b-1}} \cdot |\mathcal{T}|^{b^d}, \tag{7}$$

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 full tree of depth $d$ and branching factor $b$ is the same as the number of nonterminal nodes plus terminal nodes minus one (root node):

$$b^1 + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - b}{b-1}. \tag{8}$$

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:

$$\frac{b^{d+1} - b}{b-1} + \frac{b^d - 1}{b-1} = \frac{b^{d+1} + b^d - b - 1}{b-1}. \tag{9}$$

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

$$|\mathcal{W}| = |\mathcal{V}|^{\frac{b^{d+1}+b^d-b-1}{b-1}} \tag{10}$$

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 or simplistic heuristic search methods. This motivates us to use a principled method for evolving neural trees.

## 4. Theoretical foundation

In this section we present the theory and general principles for evolutionary computation for designing and evolving neural trees. This section aims to provide an outline of the Bayesian approach and the algorithms are detailed in the following section.

In the Bayesian approach to neural tree evolution, the *best* neural tree is defined as the *most probable* model of the data, given the data $D$ plus the prior knowledge on the problem domain. Bayes theorem provides a direct method for calculating such probabilities [10]. It states that the posterior (i.e. after observing the data $D$) probability of a neural tree $A$ is

$$\begin{aligned} P(A|D) &= \frac{P(D|A)P(A)}{P(D)} \\ &= \frac{P(D|A)P(A)}{\int_{\mathcal{A}} P(D|A)P(A)dA}, \end{aligned} \tag{11}$$

where $\mathcal{A}$ is the space of *all* possible neural trees (in case of $A$ taking discrete values, the integral will be replaced by summation). Here $P(A)$ is the prior (i.e. before observing the data) probability distribution for the neural trees, and $P(D|A)$ is the likelihood of the neural tree for the data.

The objective of the Bayesian evolutionary algorithm is to find a neural tree $A^g_{best}$ that maximizes the posterior probability:

$$A^g_{best} = \min_{g \leqslant g_{max}} \arg\max_{A^g_i \in \mathcal{A}(g)} P_g(A^g_i|D), \tag{12}$$

where $g_{max}$ is the maximum number of generations, $A^g_i$ is the $i$th neural tree at generation $g$, and $\mathcal{A}(g)$ is the population of size $M$:

$$\mathcal{A}(g) = \{A^g_i,\ i = 1, \ldots, M\}. \tag{13}$$

The posterior probability $P_g(A^g_i|D)$ of neural tree $A^g_i$ is computed with respect to the $g$th population:

$$P_g(A^g_i|D) = \frac{P(D|A^g_i)P(A^g_i)}{\sum_{j=1}^{M} P(D|A^g_j)P(A^g_j)}, \tag{14}$$

where $P(D|A^g_i)$ is the likelihood and $P(A^g_i)$ is the prior probability of (or degree of belief in) $A^g_i$. Note that the posterior probability is approximated by a fixed-size population $\mathcal{A}(g)$ which is typically a small subset of the entire neural tree space $\mathcal{A}$.

Variation operators are applied to generate $L$ offspring $A'_k$, $k = 1, \ldots, L$. Formally, this proceeds in two steps. First, candidates are generated by sampling from the proposal distribution

$$Q_g(A'_k|A^g_i) \tag{15}$$

The specific form of $Q_g(\cdot|\cdot)$ is determined by the application. Then, each candidate generated by genetic operators is accepted with probability

$$a_g(A'_k|A^g_i) = \min\left\{1, \frac{P_g(A'_k|D)}{P_g(A^g_i|D)}\right\}, \tag{16}$$

where $P_g(A^g_i|D)$ is computed by Eq. (14) and $P_g(A'_k|D)$ is the posterior probability of $A'_k$ estimated with respect to the current population:

$$P_g(A'_k|D) = \frac{P(D|A'_k)P(A'_k)}{\sum_{j=1}^{M} P(D|A^g_j)P(A^g_j)}. \tag{17}$$

If $A'_k$ is rejected in Eq. (16), then $A^g_i$ is retained, i.e., $A'_k \leftarrow A^g_i$. Note that this acceptance function does not exclude the case that $A'_k$ is generated by crossover from $A^g_i$ and another parent $A^g_j \in \mathcal{A}(g), j \neq i$. Optionally, the sampling process can be applied multiple times to

1. Initialize $M$ neural trees $A_i^1$, $i = 1, ..., M$, of heterogeneous units . Initialize $D$ of $N$ training cases. Set $g \leftarrow 1$.
2. Compute fitness values $F_i(g) = E_i(g) + \alpha(g)C_i(g)$, $i = 1, ..., M$.
3. Compute posterior probabilities $P_g(A_i^g|D)$, $i = 1, ..., M$, by Equation (??).
4. Generate $L$ offspring $A'_k, k = 1, ..., L$, by sampling from $P_g(A_i^g|D)$ using variation operators.
5. Select $M$ parents $A_i^{g+1}, i = 1, ..., M$, of generation $g + 1$ from $A'_k, k = 1, ..., L$.
6. Set $g \leftarrow g + 1$. If $g \leq g_{max}$, go to step 2.

Fig. 2. Outline of the Bayesian evolutionary algorithm for evolving heterogeneous neural trees.

make a local search in the neighborhood of the current search point.

After $L$ offspring $A'_k, k = 1, \ldots, L$, are generated, $M$ of them are selected to build the new population:

$$\mathcal{A}(g+1) = \{A_i^{g+1},\ i = 1, \ldots, M\}. \tag{18}$$

This defines the posterior distribution $P_{g+1}(A_i^g|D)$ at the next generation. It should be mentioned that this formulation of offspring selection is intentionally very general so that it can accommodate various forms of existing selection schemes, such as $(\mu, \lambda)$ selection [3, 24].

In effect, the evolutionary inference step from generation $g$ to $g + 1$ is considered to induce a new fitness distribution $P_{g+1}(A_i^g|D)$ from priors $P(A_i^g)$ through posterior distribution $P_g(A_i^g|D)$ following Bayes formula, using genetic operators. Based on this theoretical framework we present in the following section implementational details on how to learn neural tree structures and how to measure their fitness values to guide the evolutionary process.

## 5. Evolving heterogeneous neural trees

### 5.1. General procedure

To find the best architecture and connection weights of neural trees consisting of various types of units, we maintain a population $\mathcal{A}$ of individuals $A_i$ at $g$th generation

$$\mathcal{A}(g) = \{A_1, A_2, \ldots, A_M\}, \tag{19}$$

where $M$ is the population size. The general procedure for evolving heterogeneous neural trees is summarized in Fig. 2. Initially, $M$ neural trees $A_i^1$, $i = 1, \ldots, M$, are created. Their raw fitness values are measured (more details below). Based on these, the posterior probabilities $P_g(A_i^g|D)$, $i = 1, \ldots, M$, are measured and used as fitness values. Then, $L$ offspring neural trees $A'_k, k = 1, \ldots, L$, are generated by sampling from $P_g(A_i^g|D)$ using variation operators. From these, $M$ neural trees $A_i^{g+1}, i = 1, \ldots, M$, are selected to constitute the population of generation $g + 1$. The whole process is repeated until the termination condition is met. Optionally, the algorithm can be extended to adopt the incremental data inheritance mechanism [35]. In this case, the training data set $D$ starts small and then is increased as generation goes on (more details in Section 5.4). The rationale behind this incremental approach is that the use of data subsets accelerates the evolution since it can save the effective time for fitness evaluation.

The raw fitness of the neural trees $A_i^g$ at generation $g$ is defined as

$$F_i(g) = E(D|A_i^g) + \alpha(g)C(A_i^g), \tag{20}$$

where $E(D|A_i^g)$ and $C(A_i^g))$ are the error and complexity of the neural tree, computed as

$$E(D|A_i^g) = \frac{1}{N}\sum_{c=1}^{N}\left(\mathbf{y}_c - f_A(\mathbf{x}_c)\right)^2 \tag{21}$$

$$C(A_i^g) = W(A_i^g) + U(A_i^g) + L(A_i^g). \tag{22}$$

Here, $N$ is the size of training set $D = \{(\mathbf{x}_c, \mathbf{y}_c)\}_{i=1}^N$ and $W(A_i^g)$, $U(A_i^g)$, and $L(A_i^g)$ denote respectively the number of weights, units, and layers. Note that the parameter $\alpha(g)$ is a function of generation $g$ and balances the two terms.

This fitness measure can be derived (see Appendix for more details) from the Bayesian evolutionary frame-

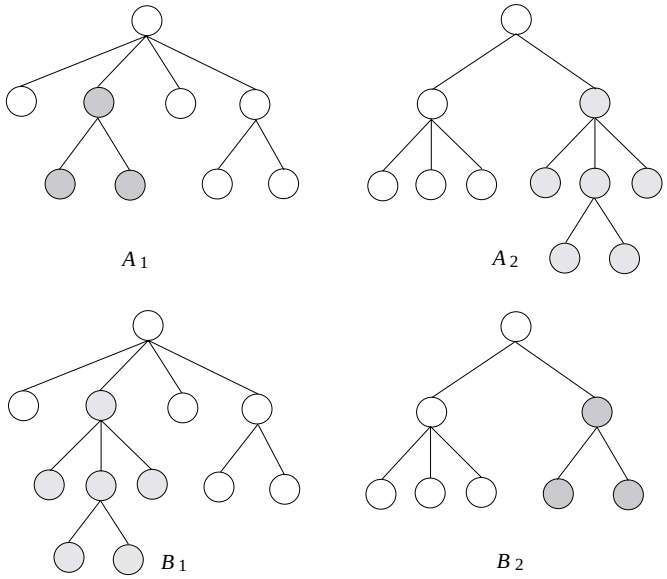


Fig. 3. Architecture adaptation by subtree crossover. The subtrees of $A_1$ and $A_2$ are exchanged to produce two offspring $B_1$ and $B_2$. During crossover, $A_1$ has grown to $B_1$ in its size and depth, and $A_2$ has shrinked to $B_2$ in its size while the depth remains the same.

work by taking the negative logarithm of the posterior probability and formulating the problem as a minimization problem. That is, under mild assumptions it can be shown that

$$\begin{aligned} F_i(g) &= -\log P(D|A_i^g) - \log P(A_i^g) \\ &= \beta E(D|A_i^g) + \alpha C(A_i^g) \\ &\propto E(D|A_i^g) + \alpha(g) C(A_i^g), \end{aligned} \tag{23}$$

where the two parameters $\beta$ and $\alpha$ are absorbed into a single parameter $\alpha(g)$. The adaptive Occam method [37] can be used to adapt the parameter $\alpha(g)$ during the run. This method distinguishes two adaptation phases. In the phase of $E_{best}(g-1) > \epsilon$, the $\alpha(g)$ decreases as the training error falls since $E_{best}(g-1) \leqslant 1$. Here, $\epsilon$ is a user-specified parameter. This encourages fast error reduction at the early stages of evolution. For $E_{best}(g-1) \leqslant \epsilon$, 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.

Note from Eqs (14) and (23) that the posterior probability $P_g(A_i^g|D)$ can be computed from the raw fitness values $F_i(g)$ as follows:

$$\begin{aligned} P_g(A_i^g|D) &= \frac{P(D|A_i^g)P_{g-1}(A_i^g)}{\sum_{j=1}^M P(D|A_j^g)P_{g-1}(A_j^g)} \\ &= \frac{\exp(-E_i(g) - \alpha(g)C_i(g))}{\sum_{j=1}^M \exp(-E_j(g) - \alpha(g)C_j(g))}, \end{aligned} \tag{24}$$

where we used symbol $P_{g-1}(A_i^g)$ to explicitly denote that the prior probability is adapted. From this equation, we see that the complexity term plays the role of the prior probability. This shows that the update of $\alpha(g)$ in the adaptive Occam method has the effect of revising the priors $P_{g-1}(A_i^g)$ since the complexity involves the revision of $\alpha(g)$.

### 5.2. Structure optimization

Structure of neural trees are modified by crossover and mutation operators. Crossover is performed by exchanging subtrees of parent trees as shown in Fig. 3. Note that crossover results in the change of the topology, size, depth, and shape of neural networks. Because of the closure property of the neural tree representation, no syntactic restriction is necessary in choosing the crossover points.

Instead of producing two offspring, one may 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 [30] and references therein). The local fitness is measured as a combination of the local error and size of the subtree, similar to the global fitness Eq. (20). The use of the size term biases evolution to choose smaller building blocks against complex ones.

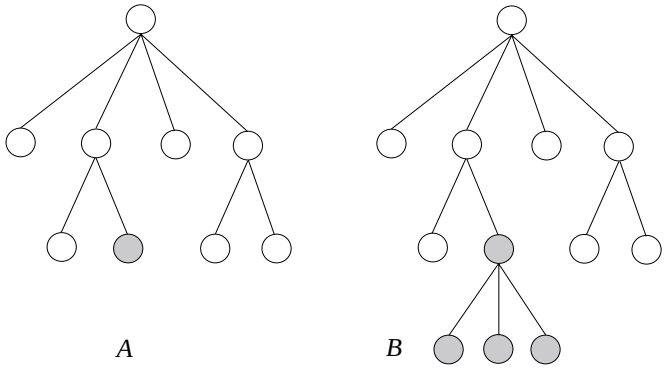


Fig. 4. Architecture adaptation (growth) by subtree mutation. In this particular example, a single node in neural tree $A$ is replaced by a subtree of size 4 generated at random to generate an offspring $B$.

Several heuristics can be applied to make the architecture adaptation more "intelligent". For example, building blocks (i.e. useful subtrees) in the population can be discovered and reused during a run. The fitness of subtrees is computated and if it exceeds a threshold, then the subtree is added to the library of building blocks. 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.

Four 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 subtrees as a whole (Fig. 4). To do this, a node is chosen at random and its subtree is replaced by a new subtree that is generated at random. Finally, mutations can be used to adapt the weight values. This is applied in conjunction with local search for weight adaptation as described in the next subsection.

### *5.3. Weight optimization*

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. 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. Each iteration applies only parametric mutation that perturbs the weight vector $\mathbf{w}$ of $A_i$ with exponential noise, a method used by BGA [24]. 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 algorithm is summarized as follows.

**Algorithm 5.1 (Local search)**

1. Given a neural tree $A$ with weight vector $\mathbf{w}^t$.
2. Generate a new weight vector $\mathbf{w}'$ by weight mutation.
3. Take as the weight vector of $(t+1)$-th step:

$$\mathbf{w}^{t+1} = \begin{cases} \mathbf{w}' & \text{w.p. } \min\left\{\frac{P(\mathbf{w}'|D)}{P(\mathbf{w}^t|D)}, 1\right\} \\ \mathbf{w}^t & \text{otherwise.} \end{cases} \quad (25)$$

4. If the termination condition not met, set $t \leftarrow t+1$ and go to Step 2.

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 $\mu_{\text{weight}}$. 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:

$$w_i' = w_i \pm R_i \cdot \delta, \quad \text{with } \delta = 2^{-K\cdot\eta} \text{ and } \eta \in [0,1]. \tag{26}$$

The random number $\eta$ 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:

$$\phi(z) = \frac{1}{K \cdot \ln(2) \cdot z} \tag{27}$$

$$\Phi(z) = Pr(Z \leqslant z) = \int_{2^{-K}}^{z} \phi(u)du = 1 + \frac{1}{K \cdot \ln(2)} \cdot \ln(z) \tag{28}$$

The exponential characteristic of $\delta = 2^{-K\cdot\eta} \in [2^{-K}, 1]$ ensures that the mutation step size of maximum value $R_i$ 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 in 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.

### *5.4. Data optimization*

The importance of training data in learning neural networks is well known. Selecting a small but representative subsets of given training data can improve the learning speed and solution quality [25]. Similar idea has also been used in evolutionary algorithms. For example, Gathercole and Ross [9] show that fitness evaluation in evolving individuals can be significantly accelerated by selecting a subset of fitness cases. The basic idea is that, other things being equal, the evolution time can be minimized by reducing the effective data size for each generation. Typically, evolution starts by considering a unique fitness case, and additional fitness cases are "gradually" taken into account when the current population meets some performance criterion. The method of incremental data inheritance [35] is a generalization of this approach.

In the evolutionary algorithm with incremental data inheritance, each individual (in our case, the neural tree) maintains its own training data set which are evolved at the same time. Just as tree structures are modified by crossover, their data sets are mixed by crossover operation on the data sets as follows.

First, two parent data sets, $D_i^g$ and $D_j^g$, are crossed to inherit their subsets to two offspring data sets, $D_i^{g+1}$ and $D_j^{g+1}$. Second, the data of parents' are mixed into a union set

$$D_{i+j}^g = D_i^g \cup D_j^g, \tag{29}$$

which is then redistributed to two offspring $D_i^{g+1}$ and $D_j^{g+1}$, where the size of offspring data sets is equal to $N_{g+1} = N_g + \lambda$, where $\lambda \geqslant 1$ is the data increment size. Thus, the size of data sets monotonically increases as generation goes on.

The diversity of the training data during inheritance is maintained by importing some portion of data from the base set. The import rate $r_i$ is given as

$$r_i = \rho \cdot (1 - d_i), \qquad 0 \leqslant \rho \leqslant 1. \tag{30}$$

where $\rho$ is a constant for import strength. The diversity $d_i$ is measured as the ratio of distinctive examples in the union set:

$$d_i = \frac{|D_{i+j}^g|}{|D_i^g|} - 1, \qquad 0 \leqslant d_i \leqslant 1. \tag{31}$$

Though the exact factor of speed-up depends on the availability of the data and its characteristics, our previous work has shown a speed-up of factor 10 by using incremental data selection in evolving Lisp-like symbolic programs [35].

## 6. Simulation results

Experiments have been carried out to compare the predictive accuracy of neural trees evolved by the Bayesian evolutionary algorithms with the performance of standard neural network solutions. For an objective comparison, we have chosen the benchmark problems collected by Prechelt [26]. Table 1 shows in the fourth column the results of the multilayer perceptrons (MLPs) as reported in [26]. The labels in the second column (for example, cancer1, cancer2 and cancer3) denote three data sets generated from the original data (Cancer data). The third column of the table shows the results of evolutionary neural trees (ENTs) for the same data sets. For each specific data set, a total of 10 runs were made. Out of 12 data sets (3 data sets for

Table 1
Performance comparison of evolutionary neural trees (ENTs) and multilayer perceptrons (MLPs). The figures are predictive errors, i.e., misclassification rate measured on the test data. The labels in the second column (for example, cancer1, cancer2 and cancer3) denote three different data sets generated from the original data (Cancer data) by Prechelt [26]. The values for MLPs are as given in [26] and the values for ENTs are averages over ten runs

| Problem | Data set | ENT | MLP |
|---|---|---|---|
| Cancer | cancer1 | 1.714 | 1.149 |
| | cancer2 | 4.381 | 5.747 |
| | cancer3 | 3.741 | 2.299 |
| Diabetes | diabetes1 | 23.95 | 25.00 |
| | diabetes2 | 23.44 | 23.44 |
| | diabetes3 | 21.12 | 21.35 |
| Heart | heart1 | 17.94 | 20.00 |
| | heart2 | 18.07 | 14.78 |
| | heart3 | 22.48 | 23.91 |
| Credit | credit1 | 14.75 | 13.95 |
| | credit2 | 16.19 | 18.02 |
| | credit3 | 17.34 | 18.02 |

Table 2
Comparison of predictive error (in terms of $t$-test) of evolutionary neural trees (ENTs) against multilayer perceptrons (MLPs). The negative $t$-values mean better performance of ENTs compared to MLPs. The larger the absolute value, the more significant the relative performance

| Cancer | Diabetes | Heart | Credit |
|---|---|---|---|
| 0.2576 | $-1.3389$ | $-0.0395$ | $-0.7488$ |

each problem and 4 problems), ENTs achieved 8 times better results than MLPs.

As the $t$-test result in Table 2 shows, evolutionary neural trees (ENTs) achieved better results in predictive accuracy than multilayer perceptrons (MLPs) for three problems out of four. ENTs most significantly outperform MLPs for the diabetes and credit problems which are known as difficult for other classifiers. The results by MLPs shown here are the performances which were achieved by Prechelt using pre-optimized model structures. Therefore, the ENT results achieved can be evaluated as very competitive to the best performance achieved by well-engineered conventional neural network methods. The predictive error for the diabetes problem seems big, in its absolute value. Our comparative analysis shows that this problem is itself very noisy and difficult for other classifiers as well, including MLPs. We found that ENTs' performance is still better than other methods. In terms of computation time, it should be mentioned that the evolutionary methods took more time than the backpropagation-trained MLPs. The additional time is well worthy of when conventional learning methods are not powerful enough to find appropriate network structures.

One distinguishing feature of evolutionary search from conventional learning algorithms for neural networks is the use of population. Maintaining multiple neural trees in a population seems useful since various subtrees of potential utility can be reserved. However, if the population size is too big, the evolutionary process is slowed down due to large requirements of resources in memory and computing time.

Another important issue in evolving neural networks is the fact that the amount of information exchanged between individuals may affect the solution quality and speed of the evolutionary design process. For example, exchange of large subtrees in crossover makes large steps in the neural architecture space, producing radically new architectures. Thus, this is a search operator of an "explorative" nature. However, in terms of stabilization, crossover of small size subtrees (i.e. "exploitative") might be more useful.

To analyze the effect of the population size and the size of subtrees for variation, we evolved neural trees using Bayesian evolutionary algorithms with varying parameter values. The problem is the prediction of laser intensity fluctuations. The data was generated from far-infrared $NH_3$ laser in a physics laboratory [15]. This problem was used as a benchmark in the 1992 Santa Fe time series competition. We considered two different classes of BEAs. In the first class, each individual is evolved by mutation only (both subtree mutation and weight mutations). Here we use a population of multiple individuals, but each individual evolves separately. We will refer to this as individual-based BEAs (iBEAs). In the second class of BEAs, the individuals evolve as in standard evolutionary algorithms both by subtree crossover and by mutation (subtree mutation and weight mutation). This will be referred to as population-based BEAs (pBEAs). In each class, we varied the size of subtrees in crossover and/or mutation.

Figure 5 summarizes the relationship between the population size and the size of subtrees for crossover and mutation. In the figure, iBEA0 (iBEA1 and iBEA2, respectively) denotes the individual-based BEA with mutation of subtree depth 0 (1 and 2, respectively), where 0 denotes mutation with addition/deletion of single branches without changing its depth. Similarly, pBEA1 (iBEA1 and iBEAm, respectively) denotes the population-based BEA with variation of subtree depth upto 1 (2 and m, respectively), where m denotes the height (maximum depth) of the tree. The algorithm iBEA0 is the most exploitative of the three individual-based BEAs, while pBEAm is the most explorative of the population-based BEAs. The experiments have been performed by varying the population size.

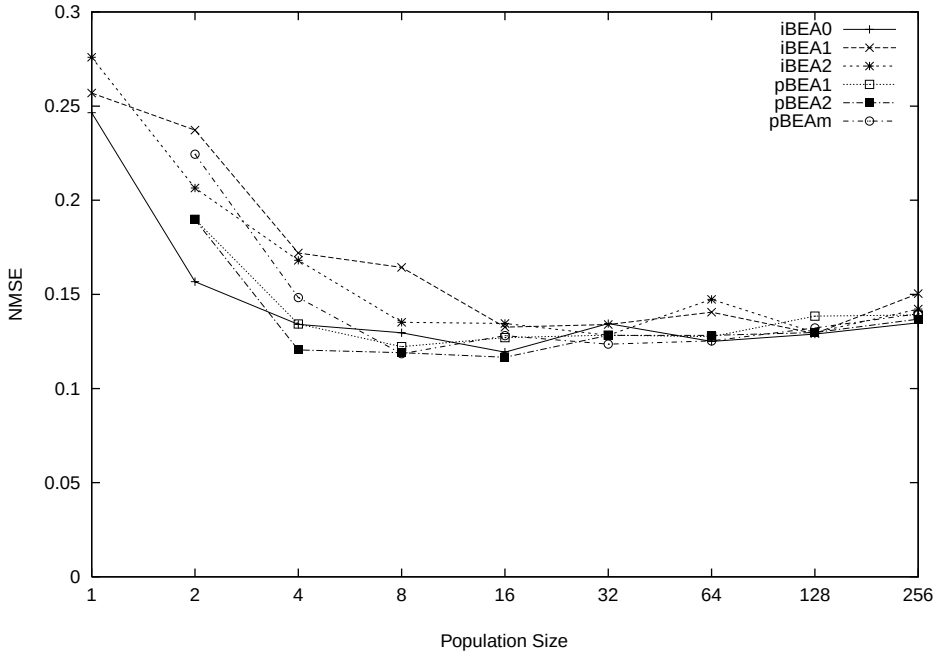


Fig. 5. NMSE values for the laser data. This figure compares the performances of the three individual-based BEAs and the three population-based BEAs as a function of the population size. The results are averaged over 10 runs. iBEA0 (iBEA1 and iBEA2, respectively) denotes the iBEA with subtree depth of 0 (1 and 2, respectively), where 0 denotes mutation with addition/deletion of single branches without changing its depth. pBEA1 (iBEA1 and iBEAm, respectively) denotes the pBEA with exchange of subtrees of depth upto 1 (2 and m, respectively), where m denotes the height or the maximum depth of the tree. It can be seen that the population-based algorithms generally perform better than the individual-based algorithms for a broad range of population sizes. The figure also shows that the individual-based algorithm with small-step mutations (iBEA0) performs best for very small ($M < 4$) populations (though its absolute performance is lower than population-based BEAs with large population sizes).

Every point in the graphs represents an average value for 10 runs. The result says that, in small populations of neural trees, variation of small subtrees is generally preferable to large ones. This is true of individual-based (mutation-oriented) variations (iBEAs) as well as population-based (recombination-intensive) variations (pBEAs). The results also show that the size of subtrees exchanged significantly affects the performance of the evolutionary neural trees when a small population (16 or less in these experiments) is used. In contrast, the size of subtrees exchanged plays a less role when they are evolved in a large population (16 or bigger in these experiments). On the other hand, the comparison of the performance curves of individual-based BEAs with those of population-based BEAs indicates that population-based evolution is in general better. The exception is when the population size is too small, i.e. $M \leqslant 4$ in these experiments, for which individual-based BEAs perform well in evolving neural trees.

Finally, it is interesting to see the shape of neural trees actually evolved by the Bayesian evolutionary algorithms. Figure 6 shows an example neural tree evolved for the laser problem. The tree contains two different types of neurons (7 summation units and one product unit) and each unit has a different number of inputs, which is contrasted with conventional neural network architectures. It should also be noted that the pi unit has three $x_1$'s as its input. This effectively represents a polynomial $x_1^3$ of degree 3 which is impossible to directly represent in multilayer perceptrons. Discovery and reuse of such partial structures (building blocks) are one of the most interesting features that distinguish evolutionary algorithms from other conventional learning algorithms for neural networks.

## 7. Concluding remarks

Optimization of neural networks for particular applications is important because the learning speed and solution accuracy are dependent on the network architecture, i.e. the type and number of units and connections, and the connectivity of units as well as the

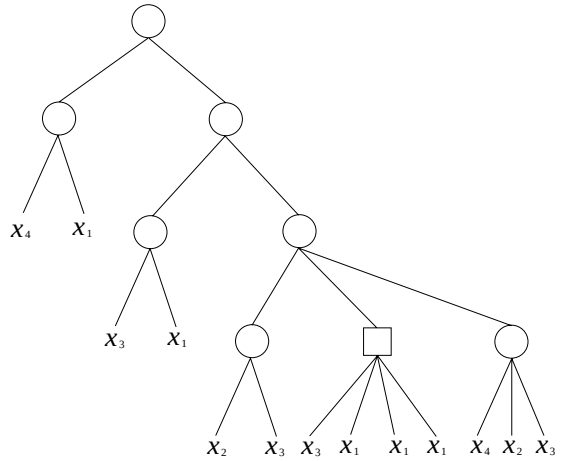


Fig. 6. A heterogeneous neural tree evolved for the time series prediction problem. This tree contains 7 summation units (circles) and one product unit (rectangle). The nodes have a variable number of inputs (variable receptive fields). Some units have duplicated inputs. In the case of the pi unit, the three $x_1$'s represent a polynomial $x_1^3$ of degree 3 which is not possible to be directly represented in multilayer perceptrons.

weights. This paper demonstrates the effectiveness of evolutionary computation for constructing application-specific neural networks of non-conventional architecture. In our experiments on a suite of benchmark problems, the heterogeneous neural trees constructed by the Bayesian evolutionary algorithms outperformed multilayer perceptrons 8 times out of 12 data sets. Analysis of the evolved solutions showed interesting structures that seem to be appropriate internal representations to solve the specific problem.

This work also demonstrates the importance of representation and fitness functions for successful application of evolutionary algorithms in the design and training of novel neural networks. The enormous size of the search space prohibits any straightforward evolutionary methods from successful application. However, appropriate representations combined with appropriately chosen fitness functions can transform the original, difficult problem to a more manageable one. Our experimental results indirectly show that neural trees are an effective representation for evolving a general class of feedforward architectures. In particular, the tree structure seems natural for the discovery of non-trivial building blocks and for the optimization of the size and shape of neural networks while not destroying important building blocks [38].

From the evolutionary computation point of view, neural networks provide an interesting problem that involves optimization of model structures, parameters, and data sets. This paper demonstrates the usefulness of the Bayesian evolutionary framework as an integrated approach to solving this problem. The Bayesian formulation of the neural network design problem allowed us to develop several principled heuristics to speed up the evolutionary process. These include the adaptive fitness functions with complexity penalty and the incremental data inheritance method. Another feature of the presented method is the use of an evolutionary algorithm for training the weights of the evolved architecture. This allows us to freely choose the class of neural architectures to evolve since evolutionary search does not make any strict assumptions on the geometry of the search space. Local search seems important in evolutionary computation that involves continuous parameter optimization, especially when it is combined with the optimization of the dimension of the parameters as in the design and training of neural networks. The concept of self-adaptation as adopted in evolution strategies seems a useful and natural refinement of the Bayesian evolutionary approach.

## Appendix: Derivation of the fitness function

For a convenient implementation of the Bayesian evolutionary algorithm we take the negative logarithm of the posterior probability $P_g(A_i^g|D)$ and use it as the (raw) fitness function

$$F_i(g) = -\log P_g(A_i^g|D), \tag{32}$$

where $A_i^g \in \mathcal{A}(g)$ and $P_g(A_i^g|D)$ is defined in its most general case by

$$P_g(A_i^g|D) = \frac{P(D|A_i^g)P(A_i^g)}{\sum_{j=1}^{M} P(D|A_j^g)P(A_j^g)}. \tag{33}$$

Then the evolutionary process is reformulated as a minimization process

$$A_{best}^g = \min_{g \leqslant g_{max}} \arg\min_{A_i^g} F_i(g), \tag{34}$$

where the fitness function is expressed as

$$F_i(g) = -\log P(D|A_i^g) - \log P(A_i^g). \tag{35}$$

Considering the exponential family of distributions, we can write the likelihood function in Bayes' theorem Eq. (11) in the form

$$P(D|A) = \frac{1}{Z_D(\beta)} \exp\left(-\beta F_D\right), \tag{36}$$

where $F_D$ is an error function, $\beta$ controls the variance of the noise, and $Z_D(\beta)$ is a normalization factor. For

example, if we assume that the data has additive zero-mean Gaussian noise, then the probability of observing a data value $y$ for a given input vector $\mathbf{x}$ would be

$$P(y|\mathbf{x}, A_i^g) = \frac{1}{Z_D(\beta)} \exp \left( -\frac{\beta}{2}(f(\mathbf{x}; A_i^g) - y)^2 \right) \tag{37}$$

where $Z_D(\beta)$ is a normalizing constant. Provided the data points are drawn independently from this distribution, we have

$$\begin{aligned} P(D|A_i^g) &= \prod_{c=1}^{N} P(y_c|\mathbf{x}_c, A_i^g) \\ &= \frac{1}{Z_D(\beta)} \exp\left(-\beta F_D\right). \end{aligned} \tag{38}$$

where $(\mathbf{x}_c, y_c) \in D$ are training cases and $F_D$ is given as

$$\begin{aligned} F_D &= E(D|A_i^g) \\ &= \frac{1}{2}\sum_{c=1}^{N}(f(\mathbf{x}_c; A_i^g) - y_c)^2. \end{aligned} \tag{39}$$

If we also assume that a Gaussian prior on the architecture of program $A_i^g$, we have

$$P(A_i^g) = \frac{1}{Z_A(\alpha)} \exp\left(-\alpha F_A\right) \tag{40}$$

where $Z_A(\alpha)$ is a normalizing constant. For example, $F_A$ can be chosen in the form

$$F_A = C(A_i^g) = \frac{1}{2}\sum_{k=1}^{K}\theta_k^2 \tag{41}$$

where $\theta_k$ are the parameters defining the neural tree $A_i^g$. This choice of prior distribution says that we expect the complexity parameters to be small rather than large, thus implementing a parsimony pressure.

Substituting Eqs (38) and (40) into Eq. (35), the fitness function is expressed as

$$\begin{aligned} F_i(g) &= \beta F_D + \alpha F_A \\ &= \beta E(D_i^g|A_i^g) + \alpha C(A_i^g). \end{aligned} \tag{42}$$

By absorbing the two parameters $\beta$ and $\alpha$ into a single parameter $\alpha(g)$, we can rewrite this as

$$F_i(g) = E(D|A_i^g) + \alpha(g)C(A_i^g) \tag{43}$$

as is given in Eq. (20).

## Acknowledgments

This research was supported by the Engineering Research Foundation of Seoul National University and by the Korea Ministry of Science and Technology through KISTEP under Grant BR-2-1-G-06 and through AITRC. Thanks to Je-Gun Joung and Dong-Yeon Cho for performing simulations.

## References

[1] P.J. Angeline and J.B. Pollack, Coevolving high-level representations, *Artificial Life III*, MIT Press, Cambridge, MA, 1993.

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

[3] T. Bäck, *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, 1996.

[4] K. Balakrishnan and V. Honavar, Evolutionary design of neural architectures, CS-TR-95-01, AI Lab, Dept. of Computer Science, Iowa State University, January 1995.

[5] R.K. Belew, J. McInerney and N.N. Schraudolph, Evolving networks: using genetic algorithm with connectionist learning, in: *Artificial Life II*, C.G. Langton et al., eds, Addison-Wesley, 1991.

[6] S.-H. Chen and C.-F. Lu, Would evolutionary computation help in designs of ANNs in forecasting foreign exchange rates? in: *Proc. 1999 Congress on Evolutionary Computation* (*CEC-99*), IEEE Press, 1999, pp. 275–280.

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

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

[9] C. Gathercole and P. Ross, Dynamic training subset selection for supervised learning in genetic programming, in: *Parallel Problem Solving from Nature III*, Y. Davidor et al., eds, Springer-Verlag, Berlin, 1994, pp. 312–321.

[10] A. Gelman, J.B. Carlin, H.S. Stern and D.B. Rubin, *Bayesian data analysis*, Chapman & Hall, London, 1995.

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

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

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

[14] G.E. Hinton and S.J. Nowlan, How learning can guide evolution, *Complex Systems* **1** (1987), 495–502.

[15] U. Hübner, C.O. Weiss, N.B. Abraham and D. Tang, Lorenz-like chaos in $NH_3$-FIR laser, in: *Time Series Prediction: Forecasting the Future and Understanding the Past*, A.S. Weigend and N.A. Gershenfeld, eds, Addison-Wesley, 1993, pp. 73–104.

[16] H. Iba and H. de Garis, Extending genetic programming with recombinative guidance, in: *Advances in Genetic Programming 2*, P.J. Angeline and K.E. Kinnear, eds, MIT Press, Cambridge, MA, 1996, pp. 69–88.

[17] A.G. Ivanknenko, Polynomial theory of complex systems, *IEEE Transactions on Systems, Man, and Cybernetics* **1**(4) (1971), 364–378.
[18] H. Kitano, Designing neural networks using genetic algorithms with graph generation system, *Complex Systems* **4** (1990), 461–476.
[19] A. Kowalczyk and H.L. Ferra, Developing higher-order networks with empirically selected units, *IEEE Trans. on Neural Networks* **5**(5) (1994), 698–711.
[20] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
[21] G.F. Miller, P.M. Todd and S.U. Hegde, Designing neural networks using genetic algorithms, in: *Proc. Third Int. Conf. on Genetic Algorithms*, Morgan Kaufmann, 1989, pp. 379–384.
[22] D. Montana and L. Davis, Training feedforward neural networks using genetic algorithms, in: *Proc. Int. Joint Conf. Artificial Intelligence*, 1989.
[23] H. Mühlenbein and J. Kindermann, The dynamics of evolution and learning – Towards genetic neural networks, in: *Connectionism in Perspective*, R. Pfeifer et al., eds, Elsevier, 1989, pp. 173–197.
[24] H. Mühlenbein and D. Schlierkamp-Voosen, The science of breeding and its application to the breeder genetic algorithm, *Evolutionary Computation* **1**(4) (1994), 335–360.
[25] M. Plutowski and H. White, Selecting concise training sets from clean data, *IEEE Trans. on Neural Networks* **4**(2) (1993), 305–318.
[26] L. Prechelt, PROBEN 1 – A set of benchmarks and benchmarking rules for neural network training algorithms, Fakultät für Informatik, Univ. Karlsruhe, Karlsruhe, Germany, Tech. Rep. 21/94, Step. 1994.
[27] A. Radi and R. Poli, Genetic programming can discover fast and general learning rules for neural networks, in: *Proc. Third Int. Conf. on Genetic Programming*, Morgan Kaufmann, 1998, pp. 314–323.
[28] D.E. Rumelhart, G.E. Hinton and R.J. Williams, Learning internal representations by error-propagation, in: *Parallel Distributed Processing*, (Vol. I), D.E. Rumelhart and J.L. McClelland, eds, MIT Press, 1986, pp. 318–362.
[29] J. Rissanen, Stochastic complexity and modeling, *The Annals of Statistics* **14** (1986), 1080–1100.
[30] J. Rosca, Towards automatic discovery of building blocks in genetic programming, in: *Proc. 1995 AAAI Fall Symposium on Genetic Programming*, J. Rosca, ed., AAAI Press, Menlo Park, CA, 1995, pp. 78–85.
[31] J.D. Schaffer, D. Whitley and L.J. Eshelman, Combinations of genetic algorithms and neural networks: A survery of the state of the art, in: *Proc. Int. Workshop on Combinations of Genetic Algorithms and Neural Networks* (*COGANN-92*), IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 1–37.
[32] D. Whitley, T. Starkweather and C. Bogart, Genetic algorithms and neural networks: Optimizing connections and connectivity, *Parallel Computing* **14** (1990), 347–361.
[33] X. Yao, Evolutionary artificial neural networks, *International Journal of Neural Systems* **4**(3) (1993), 203–222.
[34] X. Yao and Y. Liu, A new evolutionary system for evolving artificial neural networks, *IEEE Trans. on Neural Networks* **8**(3) (1997), 694–713.
[35] B.-T. Zhang, Bayesian methods for efficient genetic programming, *Genetic Programming and Evolvable Machines*, **1**(3) (2000), 217–242.
[36] B.-T. Zhang and H. Mühlenbein, Evolving optimal neural networks using genetic algorithms with Occam's Razor, *Complex Systems* **7** (1993), 199–220.
[37] B.-T. Zhang and H. Mühlenbein, Balancing accuracy and parsimony in genetic programming, *Evolutionary Computation* **3**(1) (1995), 17–38.
[38] B.-T. Zhang, P. Ohm and H. Mühlenbein, Evolutionary induction of sparse neural trees, *Evolutionary Computation* **5**(2) (1997), 213–236.
[39] B.-T. Zhang and G. Veenker, Neural networks that teach themselves through genetic discovery of novel examples, in: *Proc. IEEE Int. Joint Conf. on Neural Networks* (*IJCNN-91*), IEEE Press, 1991, pp. 690–695.