# STRIP—A Strip-Based Neural-Network Growth Algorithm for Learning Multiple-Valued Functions

Alioune Ngom, Ivan Stojmenović, and Veljko Milutinović, *Senior Member, IEEE*

***Abstract*—We consider the problem of synthesizing multiple-valued logic functions by neural networks. A genetic algorithm (GA) which finds the longest strip in $V \subseteq K^n$ is described. A strip contains points located between two parallel hyperplanes. Repeated application of GA partitions the space $V$ into certain number of strips, each of them corresponding to a hidden unit. We construct two neural networks based on these hidden units and show that they correctly compute the given but arbitrary multiple-valued function. Preliminary experimental results are presented and discussed.**

***Index Terms*—Constructive algorithm, genetic algorithm, multiple-threshold perceptron, multiple-valued logic, neural network, partitioning.**

## I. Introduction

THIS research paper proposes to synthesize multiple-valued logic functions by *minimal* multilayer feedforward neural networks. There are various measures that can be used in constructing multiple-valued neural networks. The most important measures are depth (number of layers) and size (number of processing units). The depth is related to the speed of computing a function whereas size decides the hardware cost. In this paper, we use a novel model of neuron, the multiple-valued multiple-threshold perceptron [24], [41], as basic processing element (i.e., node) of the network and attempt to implement multiple-valued logic functions by minimal number of such units. Then we study ways to compose such elements into small constant-depth networks.

The architecture of a neural network includes its topological structure, i.e., connectivity, and the transfer function of each node in the network. Architecture design is crucial in the successful application of neural networks because the architecture has significant impact on a network's information processing capabilities. Given a learning task, a neural network with only a few connections and nodes may not be able to perform the task at all due to its limited capability, while a network with a large number of nodes and connections may overfit noise in the training data and fail to have good generalization ability (also learning is slow).

Manuscript received May 27, 1999; revised December 2, 1999. The work of A. Ngom was supported by NSERC under Grant RGPIN22811700. The work of I. Stojmenović was supported by NSERC under Grant OGPIN007.

A. Ngom is with the Computer Science Department, University of Windsor, Windsor, ONN9B 3P4 Canada (e-mail: angom@cs.uwindsor.ca).

I. Stojmenović is with the Department of Computer Science, School of Information Technology and Engineering, University of Ottawa, Ottawa, ON K1N 6N5 Canada (e-mail: ivan@site.uottawa.ca).

V. Milutinović is with the Department of Computer Engineering, School of Electrical Engineering, University of Belgrade, 11120 Belgrade, Serbia, Yugoslavia (e-mail: vm@etf.bg.ac.yu).

Publisher Item Identifier S 1045-9227(01)02054-9.

Let $K = \{0, \ldots, k-1\}$ with $k > 1$. A $k$-valued logic function $f$ maps the Cartesian product $K^n$ into $K$. Denote by $P_k^n$ the set of all such functions $f : K^n \mapsto K$. Thus $P_k^n$ consists of $k^{k^n}$ multiple-valued logic functions. The set $P_k^n$ is called the set of $n$-ary operations on $K$ in universal algebras and the set of $n$-ary functions of $k$-valued logic in multiple-valued logic algebras. The set $P_k$ defined by $P_k = \bigcup_{n \geq 1} P_k^n$ is the set of all $k$-valued logic functions. For instance, $P_2$ is the set of all two-valued logic functions.

### A. Multiple-Valued Logic Neural Networks

A *discrete neuron* is a processing unit whose transfer function outputs a discrete value. An example of such transfer function is the linear threshold function. Let $R$ be the set of real numbers. A *discrete $n$-input multiple-valued neuron* has a discrete transfer function and realizes a function of $n$ variables ranging in a set $S \subset R$ with values in $K$, that is computes a function $f : S^n \mapsto K$. For $S \subseteq K$ we refer to the processing unit as a *multiple-valued logic neuron* since it simulates a multiple-valued logic function $f : K^n \mapsto K$. Our model of multiple-valued logic neuron is the *$n$-input $k$-valued $s$-threshold perceptron* defined below.

In the theory of multiple-valued logic functions there is an important class of functions called *multiple-valued multiple-threshold functions* [1]. Such functions are used in the design of classes of multiple-valued logic circuits called *programmable logic arrays* [30]. A *$k$-valued $s$-threshold function* of one variable $y \in R$ is defined as

$$g_{k,s}^{\vec{t},\vec{o}}(y) = \begin{cases} o_0 & \text{if } y < t_1 \\ o_i & \text{if } t_i \leq y < t_{i+1} \text{ for } 1 \leq i \leq s-1 \\ o_s & \text{if } t_s \leq y \end{cases} \quad (1)$$

where

$\vec{o} = (o_0, \ldots, o_s) \in K^{s+1}$ output vector;

$\vec{t} = (t_1, \ldots, t_s) \in R^s$ threshold vector with $t_i \leq t_{i+1}$ $(1 \leq i \leq s-1)$;

$s\ (1 \leq s \leq k^n - 1)$ number of threshold values.

A $n$-input $k$-valued $s$-threshold perceptron [23], [40], [24], [41], abbreviated as $(n, k, s)$-*perceptron*, computes a *weighted $n$-input $k$-valued $s$-threshold function* $F_{k,s}^n(\vec{w}, \vec{t}, \vec{o})$ given by

$$F_{k,s}^n(\vec{w}, \vec{t}, \vec{o})(\vec{x}) = g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x}) \quad (2)$$

where

$\vec{x} = (x_1, \ldots, x_n) \in K^n$ input vector;

1045–9227/01$10.00 © 2001 IEEE

$\vec{w} = (w_1, \ldots, w_n) \in R^n$ weight vector;
$\vec{w}\vec{x}$ dot product of $\vec{w}$ and $\vec{x}$.

The perceptron's *transfer function* is a $k$-valued $s$-threshold function $g_{k,s}^{\vec{t},\vec{o}} : R \mapsto K$. It is well known that any $n$-input $k$-valued logic function $f$ can be transformed into a $k$-valued $s$-threshold function, for some $s$. A $(n, k, s)$-perceptron is *homogeneous* if $s = k - 1$ and $\vec{o}$ is the identity permutation of $K$, it is *permutably homogeneous* if $s \leq k - 1$ and $\vec{o}$ is any permutation of $s + 1$ elements out of $K$. Functions computed by single such neurons are called *permutably homogeneous functions*.

A $(n, k, s)$-perceptron partitions the inputs $\vec{x}$ into $s + 1$ disjoint classes $H_0^{[o_0]}, \ldots, H_s^{[o_s]}$ using $s$ parallel hyperplanes, where $H_i^{[o_i]} = \{\vec{x} \in K^n | f(\vec{x}) = o_i \text{ and } t_i \leq \vec{w}\vec{x} < t_{i+1}\}$ (we assume $t_0 = -\infty$ and $t_{s+1} = +\infty$). Each hyperplane equation denoted by $H_i$ $(1 \leq i \leq s)$ is of the form

$$H_i : \vec{w}\vec{x} = t_i. \tag{3}$$

A $(n, k, s)$-perceptron can be decomposed into a two-hidden-layer network composed of $n$ input nodes, that is, one linear combiner in the first hidden layer, $s$ $(n, 2, 1)$-perceptrons (usual linear threshold units) in the second hidden layer and one linear combiner in the output layer [1]. The transfer function $g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x})$ of the $(n, k, s)$-perceptron is expressed as the linear summation of $s$ linear threshold functions $g(\vec{w}\vec{x})_1, \ldots, g(\vec{w}\vec{x})_s$, that is

$$g_s^{\vec{t},\vec{o}}(\vec{w}\vec{x}) = o_0 + \sum_{i=1}^{s} a_i g(\vec{w}\vec{x})_i$$

and

$$g(\vec{w}\vec{x})_i = \begin{cases} 0 & \text{if } \vec{w}\vec{x} < t_i \\ 1 & \text{if } \vec{w}\vec{x} \geq t_i \end{cases} \tag{4}$$

where $a_i = o_i - o_{i-1}$ is the weight associated with $g(\vec{w}\vec{x})_i$ and $1 \leq i \leq s$. Fig. 1 shows example of such decomposition. Fig. 2 shows the depth-two network corresponding to equation (4).

Multiple-valued logic neural networks, in our definition, are thus neural networks composed of $(n, k, s)$-perceptrons as processing units. It should be noted that the units are not necessarily the same. For example, one layer of the network may contain $(2, 4, 8)$-perceptrons while another layer may contain $(5, 4, 3)$-perceptrons.

Analog computers are inherently inaccurate due to imperfections in fabrication and fluctuations in operating temperatures. The classical solution to this problem uses extra hardware to enforce discrete behavior. However, the brain appears to compute reliably well with inaccurate components without necessarily resorting to discrete techniques. The *continuous neural network* is a computational model based upon certain observed features of the brain. Experimental evidence has shown continuous neural network to be extremely fault-tolerant; in particular, their performance does not appear to be significantly impaired when precision is limited. It has been shown by Obradović [26] that analog neurons of limited precision are essentially multiple-valued logic neurons. The first model of multiple-valued logic neuron (and neural network) was introduced by Chan [4] and since then various other models have been described (see for instance, [22], [26], [36]).

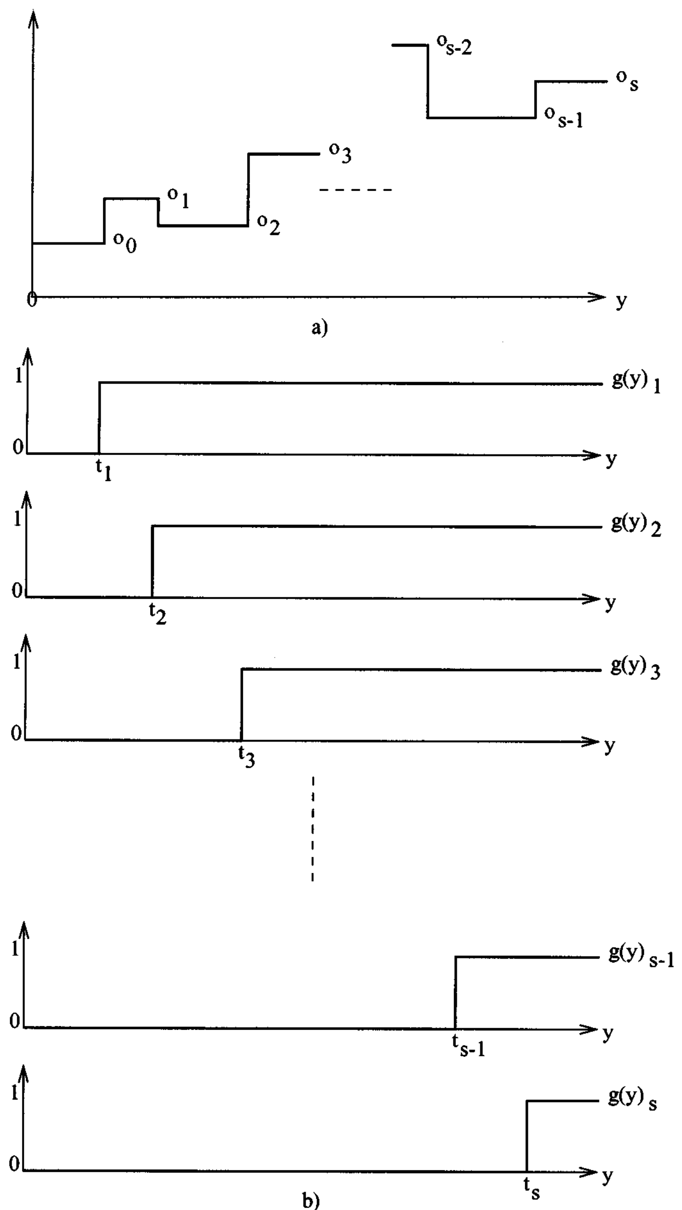


Fig. 1. Decomposition of a) $g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x})$ into b) $s$ linear threshold functions.

## *B. Problem Statement*

The problem we address in this paper is that of learning multiple-valued logic functions using minimal neural networks composed of $(n, k, s)$-perceptrons. Multilayer feedforward neural networks are in principle able to learn any arbitrary mapping, provided that enough hidden units are present [19]. For these networks, learning algorithms such as backpropagation [29] have been found to be computationally prohibitive. Also, the topology of the networks must be fixed before learning.

Until now, architecture design has been very much a human expert's job. It depends heavily on the expert experience and a tedious trial-and-error process. There is no systematic way to design a near optimal architecture for a given task automatically. Research on destructive and constructive algorithms rep-

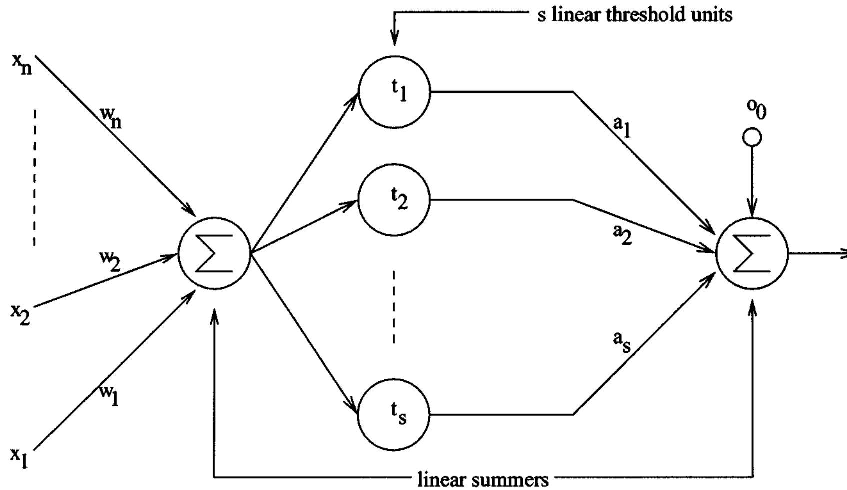


Fig. 2. Two-hidden-layers network for $g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x})$.

resents an effort toward the automatic design of architectures [38]. Design of the optimal architecture for a neural network can be formulated as a search problem in the architecture space where each point represents an architecture. Given some performance criteria about architectures, the performance level of all architectures forms a discrete surface in the space. The optimal architecture design is equivalent to finding the highest point on this surface.

A way to improve the performance of a neural network is to match its topology to a specific task (i.e., a set of input–output pairs) as closely as possible. However, the problem of deciding whether or not a given task can be performed by a given architecture is known to be *NP*-complete [13]. Also, it has been shown in [3] that the problem of finding the absolute minimal architecture for a given task is *NP*-hard. A third problem known to be *NP*-complete [34] and which also concerns us here is the following. Given two subsets $S_1,\ S_2 \subseteq R^n$ such that $|S_1 \cup S_2| = c$, determine subsets $E_1 \subseteq S_1$ and $E_2 \subseteq S_2$ such that $E_1 \cup E_2$ is of maximum cardinality, and for some $\vec{w} \in R^n$ and $t \in R$, $\vec{w}\vec{x} - t > 0$ for all $\vec{x} \in E_1$ and $\vec{w}\vec{x} - t \leq 0$ for all $\vec{x} \in E_2$. This problem is a generalized version of the *maximum cardinality problem* (that is, find a linearly separable subset of maximum cardinality). When applied to multiple-valued logic, these three problems are also *NP*-complete since they are known to be *NP*-complete in the synthesis of two-valued logic functions, which are special case of multiple-valued logic functions.

Our approach to the problem's solution is discussed in Section III. The learning method is based on the *general principle of partitioning algorithms* discussed in Section II. A partitioning algorithm seeks to construct a minimal network by partitioning the input space into classes that are as large as possible. Each class of partition is then assigned to a new hidden unit. The connections and weights of the new units are determined appropriately in such a way that the constructed network will always give the correct answer for any input. Distinct partitioning algorithms differ in the way the input space is partitioned. Also network topologies obtained from different partitioning algorithms may differ in the way new hidden units are connected.

### C. Background and Motivations

Chan [4] described three-valued logic networks called *neural logic networks* which combine the strengths of neural networks and expert systems. Such networks are used to represent a set of nonrecursive propositional rules, and to infer the truth values of the unknown propositions by applying the common backpropagation training method. Tang [35] proposed a multiple-valued logic network with functional completeness properties and learning capabilities. The multiple-valued logic network consists of layered arithmetic piecewise linear units. Since the arithmetic operations of the network are basically wired sums and piecewise linear operations, their implementation should be rather simple and straightforward. In Wang [37], neural networks composed of a novel model of multiple-valued logic neuron suitable for representing arbitrary multiple-valued logic expressions are described. Obradović [25] described algorithms for learning multiple-valued logic functions on either a single homogeneous $(n,\ k,\ k-1)$-perceptron or a depth-two network composed of $k$ $(n,\ 2,\ 1)$-perceptrons in the hidden layer and one homogeneous $(k,\ k,\ k-1)$-perceptron in the output layer. Ngom [24], [41] introduced learning algorithms for permutably homogeneous $(n,\ k,\ s)$-perceptrons and proved them to be more powerful than the algorithms mentioned in [25]. These are but a few mentioned methods for learning multiple-valued logic functions; the reader can refer to [22] for a survey on multiple-valued logic neural networks. The minimal size of a neural network for learning a given but arbitrary multiple-valued logic function has not been studied in literature. That is, the network is fixed in advance before learning and its size does not change during and after learning. As stated in Section I-B, such networks may not be powerful enough for learning or may overfit the training data. In particular, there is no evidence of any function for which the size of the learning network is significantly smaller than the size of its sum-of-products or other standard gates implementations.

Techniques are known in literature for learning multiclass functions (multiple-valued logic functions are special cases of multiclass functions). The most powerful one is the *error-correcting output codes* (or *ECOC*) approach which is a robust

method for solving multiclass learning problems. Dieterich [5] has shown that *ECOC* learning provides a general purpose method for improving the performance of inductive learning programs on multiclass problems. Other simpler and less powerful methods use backpropagation (or any appropriate learning algorithm such as madaline or radial basis function, etc.) networks with multiple output units assigned to distinct classes. All these approaches use fixed-architecture networks for learning arbitrary multiclass functions and thus a given network may not be the optimal one for such functions.

## II. Known Partitioning Methods

In Marchand [17], given a function $f \in P_2^n$, the cube $\{0, 1\}^n$ is partitioned into regions by hyperplanes in such a way that $f$ is constant in each region containing input vectors. The halfspaces defined by these hyperplanes correspond to threshold units in the hidden layer. The construction of the halfspaces implies that one can add an output unit in such a way that the resulting neural network indeed computes $f$. The hyperplanes are determined sequentially. First, an hyperplane is found such that $f$ is constant on one of its sides. The points in the corresponding halfspace are removed and they continue with the remaining points in a similar manner, until the set of remaining points becomes empty. Thus each halfspace is used to cut off a set of points with identical function values from the remaining set of points. Let the halfspaces constructed by their algorithm be $H_1, \ldots, H_r$ and define $u_i$ to be zero (or one) if $f$ is zero (or one) on the region cut off by $H_i$ for $1 \leq i \leq r$. Then adding an output unit with threshold zero and weight $u_i 2^{r-i}$ for the edge leaving the hidden unit corresponding to $H_i$, they get a neural network that computes $f$. They assume that linear threshold units produce an output zero or one. In a restricted version of their approach called *regular partitioning*, Ruján [28], the hyperplanes do not intersect. This description gives the general principle of partitioning algorithms. Particular implementations depend on the way the next hyperplane is selected and the way new hidden units are connected (in term of network weights and topology). Experiments indicate that partitioning algorithms are successful in the sense that they efficiently construct (near) minimal neural networks [14], [16], [17], [28].

Another way to view this process, without any reference to neural networks, is to consider a sequence of halfspaces $H_1, \ldots, H_r$. For each halfspace $H_i$ we specify the value $u_i$ of $f$ for those points in $H_i$ that are not contained in any of the *previous* halfspaces. Thus, for every input vector $\vec{x} \in \{0, 1\}^n$ it holds that $f(\vec{x}) = u_i$, where $i$ is the *smallest* index such that $\vec{x} \in H_i$. It is assumed that $H_r$ contains $\{0, 1\}^n$, thus $i$ is always defined. This model is called a *linear decision list* [16], [27].

More formally, Rivest [27] defines a linear decision list in the following way. A linear test $L$ over the variable $\vec{x} = (x_1, \ldots, x_n) \in \{0, 1\}^n$ is of the form $\sum_{i=1}^n w_i x_i \geq t$, where $w_1, \ldots, w_n \in R^n$ are the weights and $t \in R$ is a threshold. A linear decision list $D$ over $\vec{x}$ is a sequence $(L_1, u_1), \ldots, (L_r, u_r)$ where $L_i$ is a linear test and $u_i$ is zero or one for $1 \leq i \leq r$. It is assumed that $L_r$, the last linear test is true for all input vectors $\vec{x}$. The length of $D$ is $r$. The two-valued function $f_D$ computed by $D$ assigns to every $\vec{x}$ the value $u_i$, where $L_i$ is the *first* linear test in the list that is satisfied by $\vec{x}$. Every two-valued function $f \in P_2^n$ can be computed by some linear decision list. For instance, a disjunctive normal form with $m$ terms can be represented by a linear decision list of length $m + 1$.

Many heuristics such as Fahlman's *cascade correlation algorithm* [6], Frean's *upstart algorithm* [8], Sethi's *entropy nets* [32], Sirat's *neural trees* [33], Mezard's *tiling algorithm* [18], Barkema's *patch algorithm* [2], Frattale-Mascioli's *oil-spot algorithm* [7], Young's *carve algorithm* [39], *regular partitioning* [28], and other partitionning algorithms [9], [11], [16], [17], [20], [21], are proposed as approaches for building networks which are (near) minimal for a given arbitrary task. These heuristics are known as *constructive* or *growth* algorithms since they all construct a network starting from a fixed small number of units.

Partitioning techniques such as cascade correlation [6], upstart algorithm [8] and entropy nets [32] apply only for functions with Boolean-valued outputs. Moreover, most of the growth algoritms described in literature (see, for instance, [2], [7], [11], [17], [18], [20], [28]) are only applicable to problems with Boolean-valued inputs. Very few constructive methods deal with multiclass problems ($k$-valued functions are multiclass functions). For instance, the carve algorithm [39] and Marchand's neural decision list [16] apply both to functions $f : R^n \mapsto K$.

The techniques in [16], [39] seek to identify the largest subset of input vectors of same class that is separable from vectors of other classes. The hyperplane determined by the maximum separable subset is then assigned to a newly created $(n, 2, 1)$-perceptron. A drawback of Marchand's NDL (neural decision list) [16] is that, since it employs linear programming to determine whether specific subsets of the input vectors are linearly separable, the number of possible subsets of points that could be considered for linear separability is exponential in the size of the inputs set. In order to circumvent this problem, [16] worked only with a specific class of functions called *halfspace intersections*. Young's CARVE [39], which is an extension of Marchand [17] to multiclass functions with real-valued inputs, avoids the issue of testing for linear separability of subsets of points by directly searching for hyperplanes that separate sets of points of one class only. Let $S_i$ be a set of points of class $i$ and $H_i$ (the hull set for $S_i$) be the set of all points in the training set except those in $S_i$, $i \in K$. Clearly, points outside the convex hull formed by $H_i$ are all of the same class $i$ and thus, only points outside the convex hull can actually be separated by a hyperplane boundary from the hull set. So to find the maximum separable subset, CARVE considers only hyperplanes that touch the boundary [that is, $(n-1)$-dimensional faces or vertices] of the convex hull. The hyperplane with the largest set of points in its open halfspace outside the convex hull is the maximum separable subset. The algorithm is a simple hill-climbing technique that searches for a (near) optimal hyperplane by partially traversing (or covering) the convex hull (i.e., the boundary of the hull set) starting from a randomly selected convex hull vertex and a hyperplane that passes through that vertex. This is repeated a fixed number of times $n_v$ and each time, the initial hyperplane is randomly rotated a fixed number of times $n_r$ around the boundary of the

hull set. The set of points encountered during each rotation is determined; these are potential solutions. The main difficulty with CARVE is that the likelihood to find the maximum separable subset depends on the parameters $n_v$ and $n_r$. For larger values of the product $n_v n_r$, the more likely is one to obtain a (near) optimal hyperplane since large section of the hull set boundary will be traversed (it should be noted that the number of facets and vertices of the convex hull may be exponential in the dimension of the input space). The optimal values for $n_v$ and $n_r$ depend on the actual problem and the user must find such values by trial and error. In the end, the size of the network obtained by CARVE depends on $n_v$ and $n_r$.

In this paper we introduce a method of partitioning a set $V \subseteq K^n$ using genetic algorithm [10] to grow a multiple-valued logic neural network for learning a function. In our own approach, the maximum separable subset is treated as a special case of the longest strip (as will be discussed later). We introduce $(n, k+1, 2)$-perceptrons which will freely reduce the network size for given arbitrary $k$-valued functions.

Both CARVE and NDL use local search to search for good solutions. They extensively search some parts of the space (around faces of a convex hull as in CARVE, or, around neighborhood of separating hyperplanes as in NDL), while leaving some other parts of the search space untouched. On the other hand, GA (in our implementation) performs a global search; it treats all parts of the space equally, due to its implicit parallelism.

## III. LONGEST STRIP-BASED GROWTH ALGORITHM

In this paper, we describe a novel neural-network growth algorithm based on the search for the longest strip rather than the maximum separable subset. Our technique can be applied to various kinds of functions. That is, it is capable of handling 1) real-valued and $k$-valued inputs ($k \geq 2$) and 2) two-class problems as well as multiclass problems. In particular, we apply our growth algorithm to the synthesis of multiple-valued logic functions, that is functions of the form $f\colon K^n \mapsto K$.

A *strip* is a set of points between two parallel hyperplanes which have the same value. The *longest strip* is the strip with the maximum possible cardinality. A *maximum separable subset* is a set of points having equal values with the maximum possible cardinality that can be separated from all other points by exactly one hyperplane. Examples of longest strip and maximum separable subset are shown, respectively, in Fig. 3 and 4.

We describe a search method for finding the longest strip as well as the maximum separable subset of a currently given set of training examples, namely genetic algorithm (GA). The maximum separable subset problem is a special case of the longest strip problem. Indeed, they both share the same problem representation used by GA, but differ only by their respective objective functions and constructed network architectures. In the next paragraph we briefly describe our main growth algorithm which constructs a network by removal of points in a predefined objective subset $A \subseteq V$. $A$ is either the longest strip or the maximum separable subset in $V$. Our main objective is, using GA, to obtain a subset $G \subseteq V$ such that $|G|$ is as close as possible to $|A|$ if not equal to $|A|$ (of course we have $|G| \leq |A|$). Our maximum separable subset problem approach is an extension of

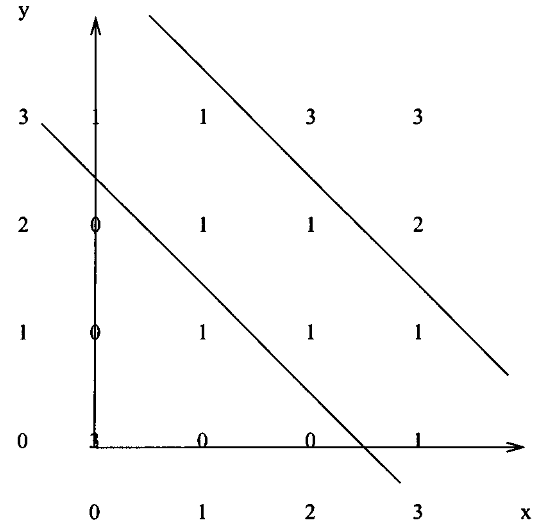


Fig. 3. Example of longest strip for $k = 4$ and $n = 2$.

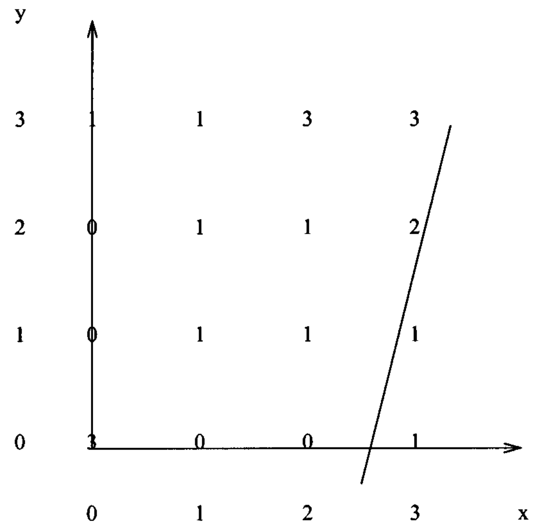


Fig. 4. Example of maximum separable subset for $k = 4$ and $n = 2$.

the sequential learning algorithm described in [17] to multiclass functions $f\colon R^n \mapsto K$. The growth algorithm of [17] is based on the perceptron learning algorithm (more specifically, on the *pocket* algorithm) and its performance was hampered by the fact that the pocket algorithm does not converge in the case where the points are not linearly separable. In our own approach, however, we use an evolutionary approach to find optimal subsets.

In our particular implementation of partitioning algorithm (see Fig. 5), GA is used to obtain subsequent halfspaces delimited by either one or two hyperplanes (depending on the predefined objective subset $A$). To each halfspace we assign a hidden unit that correctly classifies all elements of it. Let the objective subset $A$ be the longest strip in the given training set. Our growth algorithm begins with an empty first hidden layer into which new $(n, k+1, 2)$-perceptrons are inserted one after another until no more insertion is possible. A $(n, k+1, 2)$-perceptron implements two parallel hyperplanes in the input domain and the aim is to find two parallel hyperplanes that define a strip $G$ such that $|G|$ is as close as possible to $|A|$. The strip $G$ is then

**Procedure $A$-BasedSynthesis$(n, k, f)$;**

**$r := 1$;**

**$S_r := V$;**

**Repeat**

**Apply GA to find a subset $G_r$ such that $|G_r| \approx |A|$ of $S_r$;**

**Create a new hidden unit $U_r$ with respect to $A$;**

**$S_{r+1} := S_r - G_r$;**

**$r := r + 1$;**

**Until $S_r = \emptyset$;**

**Construct a network with the $r$ hidden units on the first layer;**

Fig. 5. $A$-based synthesis algorithm.

removed from the training set. The next $(n, k+1, 2)$-perceptron to be added to the network aims to separate another (near) longest strip $G$, but now only from the reduced training set. Once a (near) longest strip for this unit is found, the unit is added to the layer and the strip is removed from the training set. The construction of the first hidden layer continues with each subsequent unit separating a strip from the remaining training examples. The first hidden layer is complete when only points of one class remain in the training set. Once the first hidden layer is complete, the remaining weights, layers and units of the networks are determined to complete the network construction (the details of the network architecture are described in Section V).

Our principal objective in this paper is to synthesize any arbitrary $k$-valued logic function by a neural network constructed by our growth algorithms when (portion of) the function is given. We obtain networks that either exactly or approximately implement $k$-valued logic functions.

## IV. Determining Longest Strips by Genetic Algorithm

We use an evolutionary method to find the set of partitioning hyperplanes. Holland [12] first proposed GAs in the early 1970s as computer program to mimic the evolutionary processes in nature. Genetic algorithms manipulate a population of potential solutions to an optimization (or search) problem. Specifically, they operate on encoded representations of the solutions, equivalent to the genetic material of individuals in nature, and not directly on the solutions themselves. Holland's genetic algorithm encodes the solutions as binary *chromosome* (strings of bits). As in nature, *selection* provides the necessary driving mechanism for better solutions to survive. Each solution is associated with a *fitness value* that reflects how good or bad it is, compared with other solutions in the population. The higher the fitness value of an individual, the higher its chances of survival and reproduction and the larger its representation in the subsequent generations. Recombination of genetic material in genetic algorithms is simulated through a *crossover* mechanism that exchanges portions between two chromosomes. Another operation, *mutation*, causes sporadic and random alterations of the chromosomes. Mutation too has a direct analogy from nature and plays the role of regenerating lost genetic material and thus reopening the search.

### A. Problem Representation

Fundamental to the GA structure is the encoding mechanism for representing the problem's variables. For the problem of determining $A$, the search space is the *power set* of the training set (i.e., the set of all subsets of $K^n$). Each element of the search space is a potential solution and must be represented in such a way that meaningful and suitable genetic operators can be designed for (and applied to) it.

More formally, our population consists of chromosomes which encode the potential solutions of the problem. A potential solution is a subset $G \subseteq K^n$ and the best solution is one whose size is closest to $|A|$. Given a weight vector $\vec{w} = (w_1, \ldots, w_n) \in R^n$ we can find the strip (or separable subset) of maximum cardinality that is associated with $\vec{w}$ (see Section IV-B). For instance, the strip shown in Fig. 3 can be obtained given the vector $\vec{w} = (1, 1)$ and the function's table (moreover in this example, the obtained strip is the absolute solution). Given a training set, then searching for a subset whose size is as close to $|A|$ as possible is equivalent to searching the weight vectors space (that is $R^n$) for a vector that generates such subset. In our problem representation, a chromosome will be a weight vector $\vec{w} \in R^n$ (as in [31]) since such vector clearly encodes (i.e., represents) a potential solution to the problem. Each chromosome will uniquely determine a partition of $V \subseteq K^n$ into $s+1$ classes with $s$ parallel hyperplanes (for some $s$) and the best chromosome is the one that maximizes the number of points between a pair of parallel hyperplanes. To determine how good is a solution the GA needs an objective function to evaluate each chromosome $\vec{w}$.

We initialize the population with random real-coded chromosomes $\vec{w} \in R^n$ whose coordinates are random real numbers taken from the interval $[-1, 1]$. Each initial chromosome is then normalized to a unit vector. Another method we used for the initialization of the population is to set $w_i = \cos\alpha_i$ (for $1 \le i \le n$) for each vector $\vec{w}$, where $\alpha_i$ is a random number in the interval $[-(\pi/2), \pi/2]$. Initial population should consist of random unit hyperplanes $\vec{w}$. Starting from the initial population, we apply genetic operators such as selection, crossover, and mutation to create the next generation of chromosomes. This generational cycle is repeated until a predefined maximum number of generations is reached. Subset $G$ will be the best solution generated so far in the population.

Our main objective here is, for a given function $f$, to obtain a chromosome $\vec{w}$ which generates $G$ such that $|G| \approx |A|$. Once such $\vec{w}$ is found we create a hidden unit to be inserted in the neural network. We then eliminate all points $\vec{x}$ in $G$ and apply again the genetic algorithm on the remaining points. The algorithm terminates as soon as there are no points left. The created hidden units will then be collected to construct a feedforward network. The parameters (weight, threshold, and output vectors)

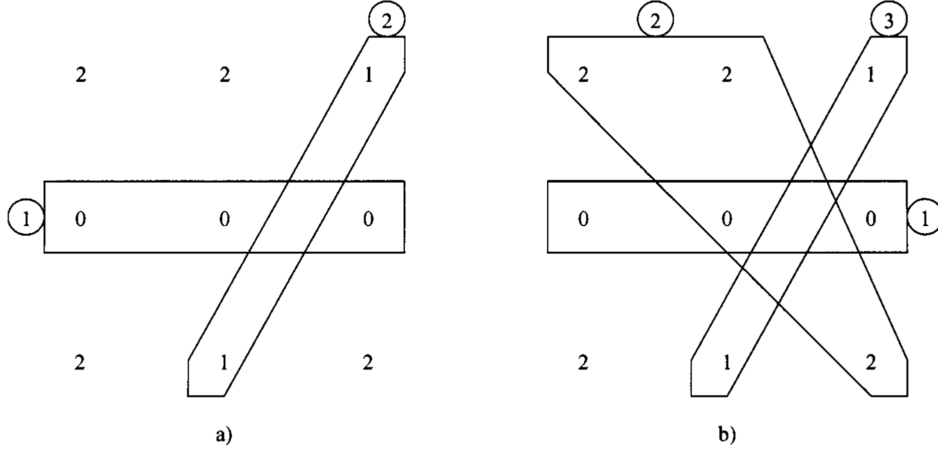


Fig. 6. Behaviors of a) $\text{Fitness2}_L$ and b) $\text{Fitness1}_L$ on some $f \in P_3^2$.

of the hidden units and the topology of the network will be discussed later.

### *B. Fitness Function*

The objective function, the function to optimize, provides the mechanism for evaluating each chromosome.

Let $S \subseteq K^n$ be the set of remaining points. Initially, $S = K^n$. To compute the longest strip generated by $\vec{w}$, we calculate for every $\vec{x} \in S$ the value $\vec{w}\vec{x}$ and construct a sorted list of records of the form $(\vec{w}\vec{x}, f(\vec{x}))$. The list is sorted using $\vec{w}\vec{x}$ as primary key and $f(\vec{x})$ as secondary key. Let these records be sorted as follows: $\vec{x}_1, \ldots, \vec{x}_{|S|}$, or more precisely, $P_i = (\vec{w}\vec{x}_i, f(\vec{x}_i))$, $1 \le i \le |S|$, where $\vec{w}\vec{x}_i \le \cdots \le \vec{w}\vec{x}_{|S|}$. A strip in $S$ is a sequence $T_{\vec{w}}^{(f(\vec{x}_i))} = P_i P_{i+1} \cdots P_{i+j}$ such that

1) $f(\vec{x}_i) = f(\vec{x}_{i+1}) = \cdots = f(\vec{x}_{i+j})$;
2) $\vec{w}\vec{x}_{i-1} \neq \vec{w}\vec{x}_i$ and $\vec{w}\vec{x}_{i+j} \neq \vec{w}\vec{x}_{i+j+1}$;

with $1 \le i \le |S|$ and $0 \le j \le |S| - i$. The length of the strip is $j - i + 1$ and $f(\vec{x}_i)$ is the value of the strip. For example, in Fig. 3 we have $\vec{w} = (1, 1)$ and

$$\begin{array}{llll} P_1 = (0, 3) & P_2 = (1, 0) & P_3 = (1, 0) & P_4 = (2, 0) \\ P_5 = (2, 0) & P_6 = (2, 1) & P_7 = (3, 1) & P_8 = (3, 1) \\ P_9 = (3, 1) & P_{10} = (3, 1) & P_{11} = (4, 1) & P_{12} = (4, 1) \\ P_{13} = (4, 1) & P_{14} = (5, 2) & P_{15} = (5, 3) & P_{16} = (6, 3) \end{array}$$

which gives the longest strip $T_{(1,1)}^{(1)} = P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13}$ generated by $\vec{w}$.

Given a set of points $S \subseteq K^n$ and a function $f$ over $S$, let $P_1 \cdots P_{j_1}$ and $P_{j_2} \cdots P_{|S|}$ $(1 \le j_1 < j_2 \le |S|)$ be, respectively, the leftmost and rightmost strips generated by $\vec{w}$, with strip values $c_1$ and $c_2$. We denote by $L(S, \vec{w})$ the length of the longest strip generated by $\vec{w}$ and denote by $M(S, \vec{w})$ the length of the maximum between the leftmost and rightmost strips, on set $S$ and function $f$. To evaluate how good is $\vec{w}$ we propose the following fitness function with respect to the definition of $A$.

- $A =$ longest strip

$$\text{Fitness1}_L(\vec{w}) = \frac{L(S, \vec{w})}{|S|}. \tag{5}$$

- $A =$ maximum separable subset

$$\text{Fitness1}_M(\vec{w}) = \frac{M(S, \vec{w})}{|S|}. \tag{6}$$

Let $S_c = \{\vec{x} \in S | f(\vec{x}) = c, c \in K\}$, that is the set of points of value $c$. An alternative objective is to select a strip of value $c$, i.e., $T_{\vec{w}}^{(c)}$, which maximizes $|T_{\vec{w}}^{(c)}|/|S_c|$, where $|T_{\vec{w}}^{(c)}|$ denotes the length of $T_{\vec{w}}^{(c)}$. That is, as in [17], [39], the selection criterion chooses the strip that constitutes the largest proportion of a class of points that can be separated. We denote by $L(S_c, \vec{w})$ the length of the largest strip proportion of value $c$ and denote by $L(S_{c_1}, \vec{w})$ and $L(S_{c_2}, \vec{w})$ the lengths of the leftmost and rightmost strips, respectively, on set $S$ and function $f$. Our alternative fitness function with respect to $A$ is

- $A =$ largest strip proportion

$$\text{Fitness2}_L(\vec{w}) = \max\left(\frac{L(S_c, \vec{w})}{|S_c|}\right) \tag{7}$$

- $A =$ largest separable proportion

$$\text{Fitness2}_M(\vec{w}) = \max\left(\frac{L(S_{c_1}, \vec{w})}{|S_{c_1}|}, \frac{L(S_{c_2}, \vec{w})}{|S_{c_2}|}\right) \tag{8}$$

where the maximum is over $S$ for every class $c$ presents in the training set. As it was stated in [39], choosing the largest proportion rather than the largest set do have some advantage for some functions $f : S \mapsto K$. For if the number of points of a class $v_1$ is small and a strip $T_{\vec{w}_1}^{(v_1)}$ constituting the whole class $S_{c_1}$ is found, it may be that a longer strip $T_{\vec{w}_2}^{(v_2)}$ with a smaller proportion $L(S_{v_2}, \vec{w})/|S_{v_2}|$ can be found. However, it is preferable to select $T_{\vec{w}_1}^{(v_1)}$ because this removes the entire set $S_{v_1}$ from $S$ and brings the neural-network construction closer to the hidden layer termination criteria of having only points of one class remaining in the training set. This is illustrated in Fig. 6 where $f \in P_3^2$ is a random function to which both fitness selection criteria were applied.

As seen in Fig. 6, it is impossible to obtain a network with exactly three hidden units (which is the absolute minimum here) when using $\text{Fitness1}_L$. The number in circles indicates the order

in which a generated strip is assigned to hidden units. So in Fig. 6b) a fourth hidden unit is needed for the last remaining point of value 2. In Fig. 6a), $Fitness2_L$ removes the set $S_1$ immediately after removing $S_0$ [unlike in b)] where a proper subset of $S_2$ is removed after $S_0$), hence one more unit is needed to remove $S_2$.

A note on the time complexity of the evaluation function. For a given $\vec{w}$, both fitness functions take $n|S|$ steps to compute the $\vec{w}\vec{x}$s, $n|S|\log|S|$ steps to sort them and at most $|S|$ steps to compute $L(S, \vec{w})$ or $M(S, \vec{w})$. Therefore the evaluation of $Fitness(1 \text{ or } 2)_{(L \text{ or } M)}(\vec{w})$ has a time complexity of $O(n|S|\log|S|)$.

Also, crossover and mutation operations below take $O(n)$ steps each and the initialization of the population takes $O(pnk^n \log k^n)$ steps ($p$ is the number of chromosomes and all initial chromosomes are evaluated for their fitness). Thus the evaluation of $Fitness(\vec{w})$ is the most expensive operation in our GA. Let $g$ be the number of generations, then at each new generation $p/2$ new chromosomes are evaluated for their fitnesses and hence, our GA has a time complexity of $O(gpn|S|\log|S|) \approx O(gpn^2k^n \log k)$.

### C. Crossover

Crossover is the GAs crucial operation. Pairs of randomly selected chromosomes are subjected to crossover. For our problem representation we propose the following mixed crossover method for real-coded chromosomes as described in [24], [41]. Let $\vec{p}_1$ and $\vec{p}_2$ be two *unit* vectors to be crossed over and let $\vec{c}_1$ and $\vec{c}_2$ be the result of their crossing. Vectors $\vec{c}_1$ and $\vec{c}_2$ are obtained using, with equal probability, two of the following three crossovers operations:

$$\vec{c}_i = \vec{p}_1 + \vec{p}_2 \tag{9}$$

$$\vec{c}_i = \vec{p}_1 - \vec{p}_2 \tag{10}$$

$$c_{i_j} = \begin{cases} p_{1_j} & \text{if random()} \le 0.5 \\ p_{2_j} & \text{otherwise.} \end{cases} \tag{11}$$

Crossover in (9) is simply the addition of two parents and the child is assured to be their exact middle vector since the parents are unit vectors. Crossover in (10) is the substraction of two parents and the child is the vector orthogonal to the sum of its parents. Crossover in (11) is a uniform crossover of two parents, that is, at coordinate $i$ each parent has 50% chances to be selected as $c_{i_j}$ $(1 \le j \le n)$. For a more efficient search, $\vec{c}_1$ and $\vec{c}_2$ must not be obtained from the same crossover operator. That is, if $\vec{c}_1$ is obtained using Formula (9) then $\vec{c}_2$ must be generated from either Formula (10) or Formula (11); this helps to maintain a certain level of diversity among chromosomes in the population. Also, crossover is applied only if a randomly generated number in the range zero to one is less than or equal to the crossover probability $p_{cros}$ (in large population, $p_{cros}$ gives the fraction of chromosomes actually crossed).

We must emphasize that each chromosome is a unit vector at any moment in the population. Thus the initial random vectors are all normalized and the childs are also normalized to unit vectors after any crossover or mutation operation.

### D. Mutation

After crossover, chromosomes are subjected to random mutations. We propose three methods of *coordinate-wise* mutations as described in [24], [41]. They correspond to bitwise mutation for binary chromosomes. Let $\vec{p}$ be a *unit* vector to be mutated to a child $\vec{c}$.

*1) Random Replacement:* With some probability of mutation, each coordinate $p_i$ $(1 \le i \le n)$ of a parent $\vec{p}$ may be replaced in the following way:

$$c_i = \text{random}[-1, 1] \tag{12}$$

where random$[-1, 1]$ returns a random real number in the interval $[-1, 1]$ with uniform probability.

*2) Orthogonal Replacement:* With some probability of mutation, each coordinate $p_i$ $(1 \le i \le n)$ of a parent $\vec{p}$ may be replaced in the following way:

$$c_i = \pm\sqrt{1 - p_i^2}. \tag{13}$$

*3) Neighborhood Replacement:* With some probability of mutation, each coordinate $p_i$ $(1 \le i \le n)$ of a parent $\vec{p}$ may be replaced in the following way:

$$c_i = p_i \pm \frac{m}{k^n} \tag{14}$$

where $m \le k$ is a random constant. Unlike the two previous methods of mutation, this method slightly rotates the current hyperplane $\vec{w}$ to a neighboring one.

Just as $p_{cros}$ controls the probability of crossover, the mutation rate $p_{muta}$ gives the probability for a given coordinate to be mutated. For a vector to be mutated, one of the three mutation operators is selected with probability $1/3$.

Here we treat mutation only as a secondary operator with the role of restoring lost genetic material or generating completely new genetic material which may be probably (near) optimal. Mutation is not a conservative operator, it is highly disruptive. Therefore we must set $p_{muta} \le 0.1$.

## V. Constructing the Neural Network

In the $r$th iteration of the $A$-based synthesis algorithm, GA will find a chromosome $\vec{w}_r$ which generates a subset $G_r$. Subset $G_r$ is the longest strip (or maximum separable subset) found in the population. The optimization ability of GA makes it possible to attain a solution as close (in size) to $|A|$ as possible, if not equal. The ability of GA to produce $|A|$ depends on its many control parameters and the complexity of the tasks to learn.

*1) Longest Strip Based Network:* At every iteration $r$ of the $A$-based synthesis algorithm, GA finds a chromosome $\vec{w}_r$ which produces the longest strip $G_r = P_i \cdots P_{i+j}$, where $1 \le i \le |S_r|$, $0 \le j \le |S_r| - i$ and strip value $v_r = f(\vec{x}_i)$. Let $u_r = v_r + 1$. Then we create a $(n, k+1, 2)$-perceptron (hidden unit $U_r$) whose weight vector is $\vec{w}_r$, threshold vector is $\vec{t}_r = (\vec{w}_r\vec{x}_i, \vec{w}_r\vec{x}_{i+j+1})$ and output vector is $\vec{o}_r = (0, u_r, 0)$. In other words, the perceptron has a transfer function of the form $g_{k+1,2}^{(\vec{w}_r\vec{x}_i, \vec{w}_r\vec{x}_{i+j+1}), (0, u_r, 0)} : R \mapsto \{0, u_r\}$ [that is a

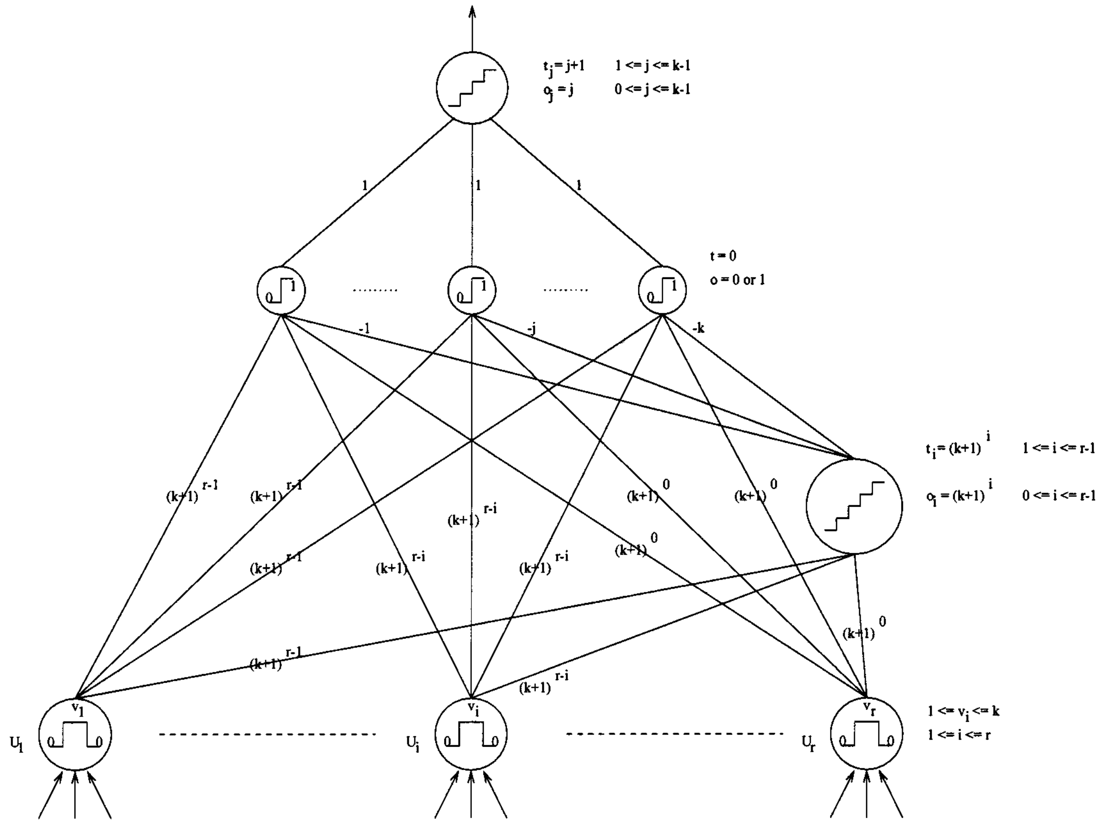


Fig. 7. Three hidden layers and $r + k + 2$ units network.

$(k+1)$-valued two-threshold function]. The $(n, k+1, 2)$-perceptron will output the value $u_r$ for all points $\vec{x} \in G_r$ and will output the value zero for all points $\vec{x} \in S_r - G_r$.

In order to achieve a good accuracy on the testing set, that is a good generalization ability of our algorithm when approximating a function, we set the threshold vector to $\vec{t_r} = (\vec{w}_r \vec{x}_i - \tau_1, \vec{w}_r \vec{x}_{i+j+1} + \tau_2)$. Thus test points of value $v_r = f(\vec{x}_i)$ which are outside but close to the strip—that is test points that lie between $\vec{w}_r \vec{x}_{i-1}$ and $\vec{w}_r \vec{x}_i$ and between $\vec{w}_r \vec{x}_{i+j+1}$ and $\vec{w}_r \vec{x}_{i+j+2}$—will be correctly classified by unit $U_r$, since they are now spanned by $U_r$. The offsets $\tau_1$ and $\tau_2$ are given by

$$\tau_1 = \frac{|(\vec{w}_r \vec{x}_i - \vec{w}_r \vec{x}_{i-1})|}{2}$$

and

$$\tau_2 = \frac{|(\vec{w}_r \vec{x}_{i+j+2} - \vec{w}_r \vec{x}_{i+j+1})|}{2}. \tag{15}$$

*2) Maximum Separable Subset Based Network:* At every iteration $r$ of the $A$-based synthesis algorithm, GA finds a chromosome $\vec{w}_r$ which produces a maximum separable subset $G_r = P_1 \dots P_{j_1}$ or $G_r = P_{j_2} \dots P_{|S_r|}$ (i.e., the maximum between the leftmost and the rightmost strips), where $1 \le j_1 < j_2 \le |S_r|$ and strip value $v_r = f(\vec{x}_1)$ or $f(\vec{x}_{j_2})$. Let $u_r = v_r + 1$. Then we create a $(n, k+1, 1)$-perceptron (hidden unit $U_r$) whose weight vector is $\vec{w}_r$, threshold vector is $\vec{t_r} = (\vec{w}_r \vec{x}_{j_1+1} + \tau_2)$ if $G_r$ is the leftmost strip, or $\vec{t_r} = (\vec{w}_r \vec{x}_{j_2} - \tau_1)$ if $G_r$ is the rightmost strip, and output vector is $\vec{o}_r = (u_r, 0)$ or $\vec{o}_r = (0, u_r)$ depending on $G_r$. In other words, the perceptron has a transfer function of the form $g_{k+1,1}^{(\vec{w}_r \vec{x}_{j_1+1} + \tau_2), (u_r, 0)}$: $R \mapsto \{0, u_r\}$ or $g_{k+1,1}^{(\vec{w}_r \vec{x}_{j_2} - \tau_1), (0, u_r)}$: $R \mapsto \{0, u_r\}$ [that is a $(k+1)$-valued one-threshold function]. The $(n, k+1, 1)$-perceptron will output the value $u_r$ for all points $\vec{x} \in G_r$ and will output the value zero for all points $\vec{x} \in S_r - G_r$. Offsets $\tau_1$ and $\tau_2$ are determined as above.

After defining all units $U_1, \dots, U_r$ (where $r$ is the number of runs of the $A$-based synthesis algorithm), the next step is to construct a feedforward multilayer neural network. We propose two network topologies.

### A. *Three Hidden Layers and* $r + k + 2$ *Units Architecture*

The network in Fig. 7 (which shows the case for strip-based method) has three hidden layers and $r + k + 2$ neurons. Hidden layer 1 contains the units (the $U_i$'s) obtained by the GA. Each unit is connected to the inputs and their parameters (weight, threshold, and output vectors) are defined as described above. So the units in this layer are either all $(n, k+1, 2)$-perceptrons or $(n, k+1, 1)$-perceptrons, depending on the definition of $A$, and there are $r$ such units (Fig. 7 shows the case $A =$ longest strip).

Hidden layer 2 has only one unit which is a $(r, (k+1)^{r-1} + 1, r-1)$-perceptron. Its weight vector $\vec{w} = ((k+1)^{r-1}, (k+1)^{r-2}, \dots, (k+1)^0)$, that is $w_i = (k+1)^{r-i}$ for $1 \le i \le r$; its threshold vector $\vec{t} = ((k+1)^1, (k+1)^2, \dots, (k+1)^{r-1})$, that is $t_i = (k+1)^i$ for $1 \le i \le r-1$; and its output vector $\vec{o} = ((k+1)^0, (k+1)^1, \dots, (k+1)^{r-1})$, that is $o_i = (k+1)^i$ for $0 \le i \le r-1$. All units of layer 1 are connected to this unit and the connection weight vector is $\vec{w}$.

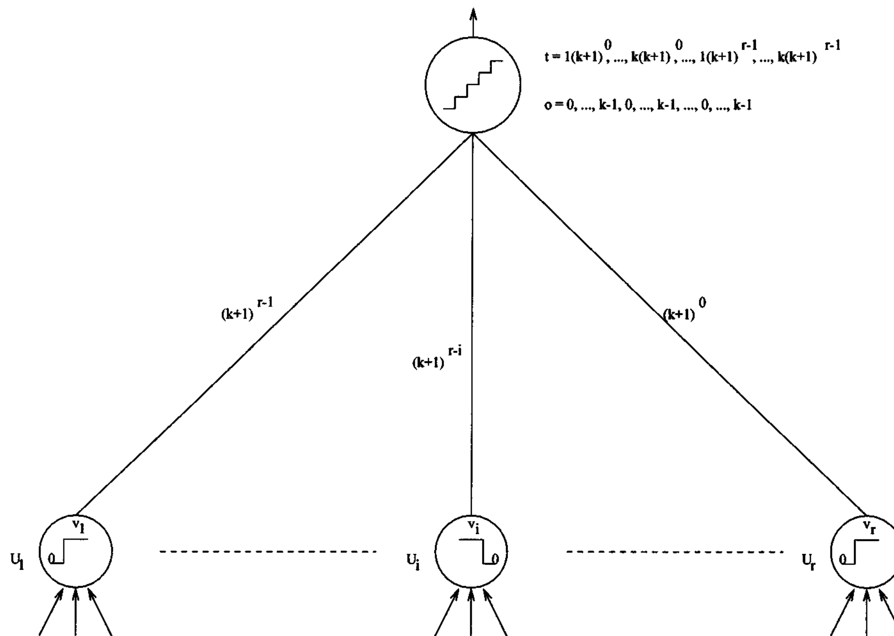


Fig. 8. One hidden layer and $r+1$ units network.

Hidden layer 3 contains $k$ units. Each unit of layer 1 and 2 is connected to every unit in this layer. Each unit is an ordinary linear threshold element [thus $\vec{o}=(0, 1)$] and the connection weight vector from layer 1 to that unit is the same as the connection weight vector from layer 1 to the unit at layer 2. The connection weight $w_{i, r+1}$ $(1 \leq i \leq k)$ from layer 2 to the $i$th unit in layer 3 is $-i$. The threshold of units in layer 3 are all set to zero.

The output layer has one unit which is a $(k, k, k-1)$-perceptron whose threshold vector $\vec{t}=(2, \ldots, k)$, that is $t_i=i+1$ for $1 \leq i \leq k-1$, and output vector $\vec{o}=(0, \ldots, k-1)$, that is $o_i=i$ for $0 \leq i \leq k-1$ (or equivalently, $o_i=t_i-1$). The connection weight from a unit in layer 3 to the output unit is one.

### B. One Hidden Layer and $r+1$ Units Architecture

The network in Fig. 8 is equivalent to Marchand's construction [17] but generalized to multiclass functions. It has one hidden layer and $r+1$ neurons. Hidden layer 1 is same as in Fig. 7 (Fig. 8 shows the case of maximum separable subsets).

The output layer has only one unit which is a $(r, k(k+1)^{r-1}+1, kr)$-perceptron. Its weight vector $\vec{w} = ((k+1)^{r-1}, (k+1)^{r-2}, \ldots, (k+1)^0)$, that is $w_i=(k+1)^{r-i}$ for $1 \leq i \leq r$; its threshold vector $\vec{t} = (1(k+1)^0, \ldots, k(k+1)^0, 1(k+1)^1, \ldots, k(k+1)^1, \ldots, 1(k+1)^{r-1}, \ldots, k(k+1)^{r-1})$; and output vector $\vec{o} = (0, \ldots, k-1, 0, \ldots, k-1, \ldots, 0, \ldots, k-1)$. All units of layer 1 are connected to this unit and the connection weight vector is $\vec{w}$.

### C. Networks Complexities

Table I shows the complexity of our constructed networks (given $f\colon R^n \mapsto K$) along with other networks that deal with multiclass functions, namely the neural decision list (NDL) network of [16] and the CARVE network of [39]. In the table, $r$ is the number of created hidden units and #Layers does not include the input layer. Also, STRIP (respectively, SEPAR) are our networks based on strips (respectively, maximum separable subsets) and the superscript $d4$ (or $d2$) specifies the architecture of Figs. 7 or 8.

TABLE I
NETWORKS COMPLEXITIES

| Networks | #Layers | #Nodes | #Thresholds | #Connections |
|---|---|---|---|---|
| SEPAR$^{d2}$ | 2 | $r+1$ | $(k+1)r$ | $(n+1)r$ |
| STRIP$^{d2}$ | 2 | $r+1$ | $(k+2)r$ | $(n+1)r$ |
| CARVE[39] | 2 | $r+k$ | $r+k$ | $(n+k)r$ |
| SEPAR$^{d4}$ | 4 | $r+k+2$ | $2r+2k-2$ | $(n+k+1)r+2k$ |
| STRIP$^{d4}$ | 4 | $r+k+2$ | $3r+2k-2$ | $(n+k+1)r+2k$ |
| NDL[16] | $r+1$ | $r+k$ | $r+k$ | $\frac{1}{2}r^2+(n+k-\frac{3}{2})r$ |

As seen from the table, STRIP$^{d2}$, CARVE and STRIP$^{d4}$ networks are all within the same order of complexities with respect to $r$, that is $O(1)$ number of layers, $O(r)$ number of nodes, thresholds and weight connections. The total number of parameters, that is the number of thresholds plus the number of weight connections, is an important complexity measure of neural networks because of their hardware cost and also their influence on generalization performance. With respect to this measure, the NDL network has the worst overall complexity, that is $O(r)$ number of layers and nodes and $O(r^2)$ number of parameters. CARVE network achieves the best overall complexity but is only slightly better than STRIP$^{d2}$ network (which has $2r-k$

more parameters but $k-1$ less nodes). STRIP$^{d4}$ network is more complex than CARVE and STRIP$^{d2}$ networks because of its asymptotic constant.

An important observation from the experiments is that, as $n$ and $k$ increase in many classes of functions, STRIP$^{d2}$ and STRIP$^{d4}$ networks have significantly much smaller values for $r$ than CARVE, NDL and SEPAR networks. If GA is used to construct a SEPAR network for a given function, then such SEPAR network should be at least smaller than a CARVE network (for the same function) for reasons explained in the last paragraph of Section II. That is, higher values of $r$ in a CARVE network are due to the less efficient search of CARVE algorithm compared to SEPAR algorithm (using GA), given the same task.

In practical applications the relation between the number of new hidden units $r$ and the target function is important. CARVE networks have the simplest operation of the basic units among them, and therefore require more new hidden units than STRIP networks. Thus, we should discuss the overall complexity with the number of new hidden units $r$ for some functions realizations. SEPAR algorithm already improves CARVE and NDL for above reasons. For most classes of functions, STRIP network implementations are significantly smaller in $r$ than maximum separable based network implementations of the same functions such as CARVE, NDL, SEPAR and other networks. The reason is that removing a strip generated by a $\vec{w}$ may create completely a new strip which is the union of two strips that enclosed the removed strip. This can happen only when the removed strip is not an end strip. For instance, consider three strips where one has value one and the other two have values zero and suppose strip of value one is between strips of values zero, that is we have the sequence 010. Then removing strip 1 creates a new strip 00, which may be longer than the strips of values one and zero together. The longer are the created strips then the smaller will be the number of new added nodes. This situation cannot happen when maximum separable subset techniques are used. That is why the maximum separable method creates more nodes than the longest strip method.

To illustrate this fact, consider the example function of $P_9^2$ (i.e., $k=9$ and $n=2$) in Fig. 9. This function has a *mirror-symmetric* table, that is all rows, columns and the two main diagonals are symmetric about their center; the second half of a row, column or diagonal is a mirror relection of the first half. Moreover, at row $y$ $(1 \leq y \leq (k-1/2))$ the first $y$ entries in that row are equal, and at column $x$ $(0 \leq x \leq (k-1/2))$ the first $x+1$ entries in that column are equal. Such two-input mirror-symmetric-table function can be constructed for any odd $k$ and the analysis is similar. This class of functions is very interesting. First, the smallest possible size for a strip based network or a maximum separable subset based network that realizes a function in this class can be obtained analytically. Second, like random functions, these functions have small separations between inputs and, therefore, seem at least *difficult* to realize. Third, as described below, they clearly demonstrate the power of STRIP compared to CARVE, NDL and SEPAR algorithms; the difference in size between a smallest STRIP network and a smallest CARVE network for a given function in this class is $O(k^2)$.

Clearly, a function in this class has a minimal representation of exactly $k$ units in the first hidden layer of STRIP. In our example function, STRIP algorithm will extract nine strips out of it and these will have values $0, 1, 2, 3, 4, 5, 6, 7, 8$ in that order. A weight vector that generates the strip of value zero (the longest strip initially) is $\vec{w}=(0, 1)$—also notice that there are four strips of values two (and lengths four) in that direction. After removal of the strip of value zero the algorithm must change direction, that is $\vec{w}=(1, 0)$, in order to remove the strip of value one. The next longest strip is the strip of value two in direction $\vec{w}=(0, 1)$; the four short strips of length four are now joined together into a single strip of length 16, since strip zero and strips one which were between them are removed.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | 6 | 4 | 2 | 0 | 2 | 4 | 6 | 8 |
| 7 | 6 | 4 | 2 | 0 | 2 | 4 | 6 | 7 |
| 5 | 5 | 4 | 2 | 0 | 2 | 4 | 5 | 5 |
| 3 | 3 | 3 | 2 | 0 | 2 | 3 | 3 | 3 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 3 | 3 | 3 | 2 | 0 | 2 | 3 | 3 | 3 |
| 5 | 5 | 4 | 2 | 0 | 2 | 4 | 5 | 5 |
| 7 | 6 | 4 | 2 | 0 | 2 | 4 | 6 | 7 |
| 8 | 6 | 4 | 2 | 0 | 2 | 4 | 6 | 8 |

Fig. 9. Mirror-symmetric-table function of $P_9^2$.

What can CARVE, NDL, and SEPAR algorithms do at best? It can be shown that the minimum number of hyperplanes needed to partition a mirror-symmetric-table function is $(k^2+2k-3)/2$. For instance, the smallest CARVE network associated with our example function contains 48 units in its first hidden layer. Clearly, the ratio between the sizes of a smallest STRIP network and a smallest CARVE network for a function in this class $\rightarrow 0$ as $k \rightarrow +\infty$. A smallest CARVE, NDL, or SEPAR network for such functions is $O(k)$ times larger than a corresponding smallest STRIP network.

Suppose for some function realization, a maximum separable subset-based algorithm, say CARVE, achieves the smallest network size $r_m$. Suppose, also for the same function realization that STRIP achieves its smallest network size $r_s$. For STRIP to be *fundamentally better* than CARVE we must have $r_s < (1/2)r_m$ (this is the case for mirror-symmetric-table functions realizations). Simply put, one STRIP neuron (a two-threshold perceptron) is equal in complexity to two CARVE neurons (one-threshold perceptrons). For functions where the inputs are separated by a number of parallel hyperplanes (such as linear, permutably homogeneous, some monotone functions), STRIP is not fundamentally better than CARVE. For these functions, a smallest STRIP network has a value $r_s \geq (1/2)r_m$. We will discuss more about this fact in Section VI.

STRIP networks are fundamentally better than CARVE, NDL, SEPAR (and other maximum separable subset based) networks for many classes of functions other than those cited in the above paragraph. More complex examples than the

mirror-symmetric-table functions can be made, for some $n$ and $k$, where STRIP selects optimal directions for partitioning whereas SEPAR, for instance, cannot *know* such information and therefore will most likely do poor job. Preliminary experiments (see Table IV, Section VI) seem to indicate that such functions (including mirror-symmetric-table functions) are *very* hard to realize by CARVE and similar algorithms; indeed, they look at least harder than random functions. For example, given the function in Fig. 9, if CARVE algorithm removes a *single* point of value six in its first few iterations, then the corresponding CARVE network will never be minimal. CARVE algorithm proceeds by *corner* (and border) separation and it is very likely that one of the points $(1, 0)$, $(1, 9)$, $(8, 0)$, $(8, 9)$ of value six will be in a singleton class that may be separated.

We can also compare the complexity and latency of a minimal STRIP network implementation of an arbitrary multiple-valued logic function to the complexity and latency of a minimal direct circuit implementation of the same function. For some functions realizations, STRIP network implementations are sometimes better and sometimes worse (in depth or size) than their corresponding direct circuit implementations. The circuit implementations can be from any basis of gates, such as *{XOR}* or *{AND, OR, NOT}* bases for example. See Section VI for a comparison with $n$-bit parity circuits.

### D. Proof of Correctness

In this section we prove that both networks, for both definitions of $A$, effectively compute any given but arbitrary function $f\colon V \subset R^n \mapsto K$.

Let $\vec{x}$ be an input vector applied to the network (we refer to both networks and both definitions of $A$). Let $\vec{u} = (u_1, \ldots, u_r)$ be the set of output values of the units of layer one when $\vec{x}$ is applied, that is unit $U_i$ outputs the value $u_i \leq k$ on input $\vec{x}$ $(1 \leq i \leq r)$. Let $p \leq (k+1)^{r-1}$ be the output of the unit (we call it $P$) of layer two. Let $\vec{q} = (q_1, \ldots, q_k)$ be the outputs of layer three, that is unit $Q_j$ has value $q_j \leq 1$ $(1 \leq j \leq k)$ on input $\vec{x}$. Let the value of the output unit (we call it $Z$) be $z \in K$.

Clearly, on input $\vec{x}$ each unit $U_i$ has either value $u_i = 0$ or $u_i = v_i \neq 0$, where $v_i$ is the maximum amplitude of the unit (see Figs. 7 and 8). Recall that each $U_i$ corresponds to a subset $G_i$ found and removed by our $A$-based synthesis algorithm during its $i$th run. The collection of the $G_i$'s is a partition of $K^n$, that is $\bigcup_{i=1}^r G_i = K^n$ and $\bigcap_{i=1}^r G_i = \emptyset$, and therefore the subset $G_i$ such that input $\vec{x} \in G_i$ corresponds to the first unit in layer one (starting from the left) which outputs a nonzero value on input $\vec{x}$. That is $\vec{u} = (0, \ldots, 0, u_i = v_i, u_{i+1}, \ldots, u_r)$, where $u_j$ for $i+1 \leq j \leq r$ is either zero or $v_j$. Recall that, according to our definition of $U_i$, $v_i - 1$ is the function value of all points in $G_i$, that is $f(\vec{x}) = v_i - 1$.

By definition, unit $P$ always outputs the value $p = (k+1)^{r-i}$ whenever $i$ is the least index in $\vec{u}$ such that $u_i > 0$ on input $\vec{x}$. In the next paragraph we let $i$ be the least such index, i.e., such that $u_i = v_i \neq 0$ and $u_1 = \cdots = u_{i-1} = 0$. Let $a = \sum_{l=1}^r (k+1)^{r-l} u_l = \sum_{l=i}^r (k+1)^{r-l} u_l$ be the dot product of the weights and inputs (i.e., the outputs of layer 1) of $P$.

Each unit $Q_j$ $(1 \leq j \leq k)$ in layer three performs the sum $b_j = a - j(k+1)^{r-i}$ (recall that the weight connection from $P$ to $Q_j$ is $-j$ and that the output of $P$ is $(k+1)^{r-i}$). We have $b_j = (k+1)^{r-i} u_i - j(k+1)^{r-i} + \sum_{l=i+1}^r (k+1)^{r-l} u_l$. From our definition of the weight connection vector between layer one and layer three it is easy to see that $0 \leq \sum_{l=i+1}^r (k+1)^{r-l} u_l < (k+1)^{r-i}$. Therefore we obtain $b_j \geq 0$ for $1 \leq j \leq u_i$ and $b_j < 0$ for $u_{i+1} \leq j \leq k$. That is, exactly $u_i$ units in layer three will output the value 1.

By definition, the output unit $Z$ which has only unit weights, will perform the sum $c = \sum_{j=1}^k 1 b_j = \sum_{j=1}^{u_i} 1 = u_i$. Since $1 \leq u_i \leq k$ and the threshold vector of $Z$ is such that $t_{u_i} = u_i + 1$, therefore we obtain $t_{u_i - 1} \leq u_i < t_{u_i}$. From the definition of the output vector of $Z$ we have that the output of the neural network is $z = t_{u_i - 1} - 1 = u_i - 1 = v_i - 1 = f(\vec{x})$. Thus the networks has effectively classified input $\vec{x}$ correctly. This completes our proof for the network of Fig. 7. The proof for Fig. 8 is straightforward and therefore omitted.

## VI. Experimental Results

In this section we present experimental results from the applications of STRIP and SEPAR algorithms to a number of multiple-valued logic functions, namely: random functions, linear functions, monotone functions, permutably homogeneous functions, $n$-bit parity functions, mirror-symmetric functions and mirror-symmetric-table functions. In analyzing the performance of STRIP on these functions we looked at two important aspects, that is the size of the networks it generates and its generalization ability. We constructed STRIP networks and compared their performances with constructed SEPAR networks along with other well-known construction techniques.

Throughout the experiments, we used the following parameters: 1000 generations for GA, 75% crossover rate, and 10% mutation rate. We also used an elitist strategy in the GA, that is the best individual of the current generation is always reproduced to the next generation. This elitist strategy helps counter-balance the disruptive effect of our high mutation rate. On the other hand we may need this high mutation rate in order to efficiently search the infinite space of weight vectors. These parameters were obtained by trial-and-error and seemed optimal.

For each given class of functions and method of network construction, we ran our algorithms ten times. All ten runs of the algorithms used the same functions (generated randomly) but with different random seeds (and different random training sets for approximation).

Table II shows, respectively, for both objective functions (the superscripts $F1$ and $F2$ stand for *Fitness*1 and *Fitness*2), the results of ten runs of each method on randomly generated functions from each of the four test classes. We display the average number of created hidden nodes, $r$ (in the first layer of the constructed networks), with its standard deviation, the minimum value found by the method (the number in parenthesis is the number of times it was found), the smallest running time (in minutes) over ten runs, and the average generalization accuracy (with its standard deviation) on the test set over ten runs. One can see from the table that, in general, *Fitness*2 yields smaller but less accurate networks than *Fitness*1.

For functions approximations (see column *Using 60% of $K^n$*) we simply used random portions of $K^n$ as training sets. Each of these training sets were generated using a different random

TABLE II
RESULTS OF 10 RUNS FOR $k = 4$ AND $n = 4$

| | | Using 100% of $K^n$ | | | Using 60% of $K^n$ | | |
|---|---|---|---|---|---|---|---|
| | | #Nodes | Min | Time | #Nodes | Time | Accuracy |
| | | $p_{muta} = 10\%$ and $p_{cros} = 75\%$ | | | | | |
| Rand. | STRIP$^{F2}$ | $28.0 \pm 0.89$ | 27 (3) | 33 | $17.6 \pm 0.66$ | 16 | $25.92\% \pm 2.42$ |
| | STRIP$^{F1}$ | $28.9 \pm 0.94$ | 27 (1) | 29 | $19.2 \pm 0.75$ | 16 | $23.11\% \pm 1.43$ |
| | SEPAR$^{F2}$ | $65.8 \pm 1.66$ | 62 (1) | 58 | $38.4 \pm 1.28$ | 29 | $25.53\% \pm 3.39$ |
| | SEPAR$^{F1}$ | $66.3 \pm 2.37$ | 63 (1) | 58 | $40.1 \pm 2.12$ | 29 | $24.95\% \pm 3.42$ |
| Line. | STRIP$^{F2}$ | $10.0 \pm 0.00$ | 10 (10) | 10 | $09.7 \pm 0.46$ | 6 | $87.28\% \pm 5.98$ |
| | STRIP$^{F1}$ | $10.0 \pm 0.00$ | 10 (10) | 10 | $09.9 \pm 0.30$ | 7 | $87.09\% \pm 3.79$ |
| | SEPAR$^{F2}$ | $20.3 \pm 3.58$ | 13 (1) | 19 | $17.8 \pm 2.56$ | 11 | $85.24\% \pm 6.78$ |
| | SEPAR$^{F1}$ | $20.2 \pm 3.60$ | 13 (1) | 19 | $17.0 \pm 2.83$ | 11 | $83.88\% \pm 4.21$ |
| Mono. | STRIP$^{F2}$ | $09.4 \pm 0.49$ | 9 (6) | 7 | $08.0 \pm 0.77$ | 3 | $80.68\% \pm 5.31$ |
| | STRIP$^{F1}$ | $10.1 \pm 0.70$ | 9 (2) | 7 | $08.3 \pm 0.46$ | 5 | $80.97\% \pm 6.41$ |
| | SEPAR$^{F2}$ | $10.6 \pm 1.28$ | 10 (4) | 7 | $09.0 \pm 1.34$ | 4 | $85.05\% \pm 6.46$ |
| | SEPAR$^{F1}$ | $12.7 \pm 1.00$ | 11 (2) | 7 | $10.8 \pm 1.08$ | 5 | $80.58\% \pm 6.22$ |
| Perm. | STRIP$^{F2}$ | $3.0 \pm 0.00$ | 3 (10) | 1 | $3.1 \pm 0.30$ | 1 | $96.99\% \pm 1.33$ |
| | STRIP$^{F1}$ | $3.1 \pm 0.30$ | 3 (9) | 4 | $3.0 \pm 0.00$ | 3 | $98.16\% \pm 2.15$ |
| | SEPAR$^{F2}$ | $3.1 \pm 0.30$ | 3 (9) | 1 | $3.1 \pm 0.30$ | 1 | $97.09\% \pm 2.46$ |
| | SEPAR$^{F1}$ | $3.3 \pm 0.90$ | 3 (9) | 5 | $3.3 \pm 0.90$ | 3 | $96.99\% \pm 1.76$ |
| | | $p_{muta} = 0\%$ and $p_{cros} = 75\%$ | | | | | |
| Rand. | STRIP$^{F2}$ | $27.9 \pm 0.94$ | 26 (1) | 28 | $17.8 \pm 0.87$ | 11 | $24.85\% \pm 3.17$ |
| Line. | STRIP$^{F2}$ | $10.0 \pm 0.00$ | 10 (10) | 13 | $09.9 \pm 0.54$ | 9 | $86.12\% \pm 3.28$ |
| Mono. | STRIP$^{F2}$ | $09.8 \pm 0.60$ | 9 (3) | 4 | $08.2 \pm 0.40$ | 3 | $82.82\% \pm 4.64$ |
| Perm. | STRIP$^{F2}$ | $03.5 \pm 0.50$ | 3 (5) | 1 | $03.2 \pm 0.40$ | 1 | $94.85\% \pm 2.64$ |
| | | $p_{muta} = 75\%$ and $p_{cros} = 0\%$ | | | | | |
| Rand. | STRIP$^{F2}$ | $27.5 \pm 1.20$ | 26 (3) | 36 | $18.6 \pm 0.80$ | 13 | $24.95\% \pm 3.16$ |
| Line. | STRIP$^{F2}$ | $10.0 \pm 0.00$ | 10 (10) | 15 | $10.1 \pm 0.30$ | 12 | $87.57\% \pm 4.27$ |
| Mono. | STRIP$^{F2}$ | $09.8 \pm 0.40$ | 9 (2) | 12 | $07.7 \pm 0.64$ | 7 | $82.14\% \pm 2.61$ |
| Perm. | STRIP$^{F2}$ | $03.0 \pm 0.00$ | 3 (10) | 1 | $03.0 \pm 0.00$ | 1 | $94.76\% \pm 2.46$ |

seed, also no two elements in the sets are equal. The network obtained with a training set is then tested on the test set, that is the remaining *40% of* $K^n$, and its accuracy on the test set is computed.

To see if there is any advantage to set $p_{muta} > 0$ or $p_{cros} > 0$ we did two experiments with STRIP using *Fitness*2, however with either $p_{muta} = 0\%$ and $p_{cros} = 75\%$ or $p_{muta} = 75\%$ and $p_{cros} = 0\%$ (see Table II). Under the column *Using 100% of* $K^n$, results in the third part of the table ($p_{cros} = 0\%$) are in general slightly better than those in the first part of the table ($p_{cros}, p_{muta} > 0$) which in turn are in general slightly better than those in the second part of the table ($p_{muta} = 0\%$). Thus there is advantage in setting $p_{muta} > 0$. Clearly, mutation is the operator that helps GA explore the search space. Crossover helps to focus the search on interesting parts of the space, in parallel. Mutation probability must be set small in order to avoid a *random walk* in space and to better *exploit* good parts of the space. Under the column *Using 60% of* $K^n$, the situation is somewhat reversed.

Random functions are more difficult to learn because of the smaller separation among their inputs. Therefore it is not surprising that they give the highest number of nodes in the tables and the lowest generalization accuracy. On the other hand, permutably homogeneous functions have larger separation of inputs which makes them easier and faster to learn. So they naturally produce smaller networks of $O(k)$ nodes on average and also give a very high generalization accuracy. Other classes of functions such as linear, monotone, and other functions lie between these two extremes.

*1) Permutably Homogeneous Functions:* Functions in this class are partitioned by at most $k - 1$ separating parallel hyperplanes such that no two distinct classes have equal values. Therefore, the minimal number of new hidden units for both STRIP and SEPAR is $\leq k$. We have experimented

with the following permutably homogeneous functions: $f(\vec{x}) = \lfloor(\sum_{i=1}^{n}(1/a_i)x_i)+n\rfloor \bmod k$, where $a_i = 2i+1$. We randomly generated the four-input four-valued logic function $f(\vec{x}) = \lfloor x_1/3 + x_2/5 + x_3/7 + x_4/9 + 4\rfloor \bmod 4$ and tested our algorithms with it. The function has three possible values, namely 0, 1, 2. The three classes are separated by two parallel hyperplanes (classes 1 being in the middle) and the distance between two adjacent hyperplanes is $\geq 2$.

The results of STRIP and SEPAR are consistent: they gave a value of about the minimal solution $r = 3$. Clearly, there is no benefit of using STRIP for these functions; there is no reduction at all and thus the STRIP network is twice more complex than its corresponding SEPAR network. In general, STRIP is not better than SEPAR for functions where the inputs are separated by *nonintersecting* hyperplanes with distinct values in distinct regions.

*2) Monotone Functions:* We experimented with a (random) function of $P_4^4$ which is monotonic under the *natural* nondecreasing order on $K$, that is $0 \leq 1 \leq \ldots \leq k-2 \leq k-1$. For such functions, STRIP and SEPAR produced closed results. This suggests that, as for permutably homogeneous functions, SEPAR is better than STRIP for these monotone functions since its achieves smaller network complexity even though STRIP has smaller values for $r$.

*3) Linear Functions:* These functions are partitioned by a number of parallel hyperplanes where, many separated distinct classes of inputs have equal values and any two adjacent classes have distinct values. The minimal STRIP network for such functions will be *exactly* twice smaller than the corresponding minimal SEPAR network, for reasons explained in Section V-C. Therefore, even though the STRIP network is twice smaller in $r$, it will have exactly the same complexity as its corresponding SEPAR network for linear functions realizations. Thus STRIP is no better than SEPAR and CARVE for such functions.

The random four-valued linear function generated was $f(\vec{x}) = (3x_1 + x_2 + 3x_3 + x_4) \bmod k$. Both STRIP and SEPAR algorithms, have difficulties realizing this function. First, the minimum $r$ produced by SEPAR, 13, is very far from the average result, 20.3. Second, STRIP should give a result which is about two times smaller than the minimum obtained by SEPAR. The reason for this is that, for linear functions, the distance between two adjacent separating parallel hyperplanes is very small ($\leq 1$). Therefore the algorithms are very sensitive to rotations, that is, small rotations from separating hyperplanes could cause the algorithms to produce large networks.

The $n$-bit parity functions are linear functions and it is well-known that a single layer minimal solution exists with $n$ hidden $(n, 2, 1)$-perceptrons. We carried out experiments with STRIP and SEPAR algorithms for $n = 0, \ldots, 12$ using 2500 generations of GA to learn such functions. The results obtained were consistently $\lceil n/2\rceil + 1$ hidden units for STRIP and $n+1$ hidden units for SEPAR, using both fitness measures respectively. For SEPAR we cannot obtain $n$ hidden units like the other maximum separable-based methods because the last training set (of points of same class) is always assigned to a new hidden unit. Thus, for instance, we will obtain exactly two hidden units for the binary *AND* function whereas the other methods would obtain one hidden unit. For SEPAR, the results obtained for $n = 10, 11, 12$ were in reality 13, 15, 18 for 2500 generations of GA, however, we obtained the correct results, 11, 12, 13 when we increased the number of generations to 4000. This suggests that our algorithms are able to find the minimal value if they are given enough time.

TABLE III
COMPARISON OF NETWORK SIZE FOR RANDOM FUNCTIONS OF $P_2^n$

| $n$ | STRIP$^{F2}$ (100) | CARVE[39] (100) | Sequential[17] (100) | Upstart[8] (100) | SEPAR$^{F2}$ (100) |
|---|---|---|---|---|---|
| 4 | $2.81 \pm 0.39$ | $2.40 \pm 0.69$ | | 3 | $3.62 \pm 0.61$ |
| 5 | $3.23 \pm 0.42$ | $3.73 \pm 0.58$ | | 4.5 | $4.93 \pm 0.50$ |
| 6 | $4.56 \pm 0.50$ | $5.88 \pm 0.67$ | $7.28 \pm 0.82$ | 8 | $6.99 \pm 0.71$ |
| 7 | $6.95 \pm 0.30$ | $9.47 \pm 0.74$ | | 16 | $11.14 \pm 0.77$ |
| 8 | $11.56 \pm 0.50$ | $16.23 \pm 0.86$ | $18.3 \pm 0.69$ | 33 | $19.51 \pm 1.03$ |

$n$-bit parity functions can be implemented by *{XOR}*-circuits of depth one and size $O(n)$, *{XOR}*-circuits of depth $O(\log_2 n)$ and size $n$, *{AND, OR}*-circuits of depth $d$ and size $O(2^{1/d-1}n^{d-2/d-1})$ or *{AND, OR, NOT}*-circuits of depth two and size $2^{n-1}$. For such functions, a STRIP$^{d2}$ network has size $\lceil n/2\rceil + 2$ and a STRIP$^{d4}$ network has size $\lceil n/2\rceil + k + 3$. As one can see, STRIP networks are not always better than their corresponding direct circuit implementations. The difference in complexity between a STRIP network and a direct circuit depends on factors such as the basis set of gates and the fan-in or fan-out of the gates in the circuit.

*4) Random Functions:* The experiments clearly show that STRIP is fundamentally better than SEPAR and CARVE for random functions realizations. The average and the minimum obtained STRIP networks are at least twice smaller than the corresponding results for SEPAR networks. As stated already, STRIP is able to change (and select good) directions for separation and to create larger classes, while SEPAR cannot do any of these two things. For random functions, the separation between classes is very small. STRIP can *see inside* a random function to decide the best directions to choose and, by doing that, it maximizes the size of all classes (they will get bigger and bigger until they can be removed). SEPAR only maximizes the size of the classes that are at the boundaries of the inputs space.

Table III contains results reported for the best three constructive algorithms (so far in literature) that have been applied to learning random two-valued logic functions. The value in parenthesis at the top of each column is the number of trials over which the network is averaged. In each trial of STRIP, we generated a different random function and used 2500 iterations of GA. STRIP produces smaller networks for this classification task than any other growth method. Significantly, STRIP gives much smaller networks than CARVE as $n$ and $k$ increase. Also, SEPAR performs better than Upstart as $n$ and $k$ increases.

*5) Mirror-Symmetric Functions:* We have carried out experiments with *mirror-symmetric functions*. A mirror-symmetric function has value one if the second half of an input vector is a mirror reflection of the first half, that is the input vector is

TABLE IV
PERFORMANCES COMPARISONS BETWEEN MIRROR-SYMMETRIC-TABLE AND RANDOM FUNCTIONS OF $P_9^2$

| | STRIP$^{F2}$ | STRIP$^{F1}$ | SEPAR$^{F2}$ | SEPAR$^{F1}$ |
|---|---|---|---|---|
| Using 100% of $K^n$ | | | | |
| Mirr. | $09.0 \pm 0.00$ | $09.8 \pm 0.40$ | $60.2 \pm 1.08$ | $69.0 \pm 3.69$ |
| Rand. | $29.2 \pm 1.99$ | $31.8 \pm 1.60$ | $58.8 \pm 2.89$ | $64.7 \pm 4.61$ |
| Using 60% of $K^n$ | | | | |
| Mirr. | $09.0 \pm 0.00$ | $11.7 \pm 1.95$ | $31.0 \pm 3.13$ | $36.0 \pm 2.49$ |
| Rand. | $19.1 \pm 1.37$ | $21.6 \pm 1.91$ | $36.1 \pm 2.98$ | $36.7 \pm 2.90$ |
| Accuracies | | | | |
| Mirr. | $65.15\% \pm 12.44$ | $57.27\% \pm 09.53$ | $18.18\% \pm 06.91$ | $13.33\% \pm 05.62$ |
| Rand. | $09.70\% \pm 04.85$ | $16.06\% \pm 06.36$ | $10.91\% \pm 07.20$ | $12.42\% \pm 05.33$ |

symmetric about its center. This function is known to have a minimal representation of two hidden units. For $n = 2, \ldots, 7$ we have always found the optimal number of hidden units in STRIP and SEPAR, two, using 1000 generations of GA. For $n = 8, \ldots, 12$ we have always found the optimum when using 3000 generations of GA. Here, as for permutably homogeneous functions, STRIP is no better than SEPAR.

*6) Mirror-Symmetric-Table (mst) Functions:* Table IV shows the comparisons of performances between our example mst function in Fig. 9 and ten random functions. For each algorithm (STRIP$^{F2}$, STRIP$^{F1}$, SEPAR$^{F2}$ and SEPAR$^{F1}$), we did ten runs on our mst function and averaged the results. Also each algorithm was applied on ten distinct random functions (different from the random functions used for the other three algorithms) and the results were then averaged. STRIP$^{F2}$ is a clear winner (except its performance in generalization for random functions); it produced the minimal size for mst. The table also shows that the single mst function is harder to realize by SEPAR than random functions (see the last two entries of the first two rows): the average result of SEPAR for mst is very far from the absolute minimum, 48, and is significantly larger than the averaged result of ten random functions. For reasons explained in Section IV-B, *Fitness*2 helps to remove the classes faster than *Fitness*1, and therefore, it yields better results in the table for both algorithms (except on generalization of random functions).

## VII. CONCLUSION

In this paper we have discussed a particular implementation of partitioning algorithm to construct (near) minimal multiple-valued neural networks for computing given but arbitrary multiple-valued functions. We used genetic algorithm to find a (near) minimal set of hidden units that partitions the space $V \subseteq K^n$. Our main contribution is indeed the design of a growth algorithm for synthesizing multiple-valued logic functions by neural networks.

STRIP is better than CARVE, NDL and SEPAR for many classes of functions in which the separations among inputs is small. Such functions include mirror-symmetric-table functions, random functions and symmetric functions. STRIP and SEPAR can be used for functions with real-valued inputs, $k$-valued inputs, two-valued outputs, or $k$-valued outputs. We have also used hill-climbing method and evolution strategy to search for longest strips or maximum separable subsets, but the results were significantly worse than those obtained by GA. More research is needed in order to increase the speed of the $A$-based synthesis algorithm.

Currently, we are investigating the generalization properties of STRIP and SEPAR to many real-world problems. We believe that our methods generalize better than CARVE since CARVE always overfits the weights. We avoided this overfitting by translating the obtained hyperplanes away from their original positions so to include some test points in the closed space delimited by these hyperplanes. Also in higher dimension and for many classes of functions, our STRIP networks are much less complex (in terms of total number of parameters) than CARVE and others maximum separable based networks, and therefore by the *Occam razor principle* STRIP would generalize better. We can further improve the generalizability by transforming an obtained STRIP network into a *radial basis functions network* and apply back-propagation to the resulting network. the authors of [39] have done the same thing with CARVE using sigmoidal units and proved it efficient. In our case, we will use radial basis functions because Gaussian units are continuous version of our $(n, k+1, 2)$-perceptrons.

## ACKNOWLEDGMENT

The authors would like to thank the referees for their important and interesting suggestions.

## REFERENCES


[1] M. H. Abd-El-Barr, S. G. Zaky, and Z. G. Vranesić, "Synthesis of multivalued multithreshold functions for ccd implementation," *IEEE Trans. Comput.*, vol. 35, no. 2, pp. 124–133, Feb. 1986.

[2] G. T. Barkema, H. M. A. Andree, and A. Tall, "The patch algorithm: Fast design of binary feedforward neural networks," *Network*, vol. 5, no. 2, pp. 393–407, 1993.

[3] A. Blum and R. L. Rivest, *Proc. 1st Workshop Comput. Learning Theory*. San Mateo, CA: Morgan Kaufmann, 1988, p. 9.

[4] L. S. Hsu, S. C. Chan, and H. H. Teh, "On neural logic networks," in *Neural Networks*. New York: Pergamon, 1988, vol. 1, p. 428.

[5] T. G. Dieterich and G. Bakiri, "Solving multiclass learning problems via error-correcting output codes," *J. Artificial Intell. Res.*, vol. 2, pp. 263–386, 1995.

[6] S. E. Fahlman and C. Lebiére, "The cascade-correlation learning architecture," *Advances Neural Inform. Processing Syst.*, vol. 2, p. 254, 1990.

[7] F. M. Frattale-Mascioli and G. Martinelli, "A constructive algorithm for binary neural networks: The oil-spot algorithm," *IEEE Trans. Neural Networks*, vol. 6, pp. 794–797, 1995.

[8] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Comput.*, vol. 2, pp. 198–209, 1990.

[9] S. I. Gallant, "Three constructive algorithms for network learning," in *Proc. 8th Annu. Conf. Cognitive Sci. Soc.*, Amherst, MA, Aug. 15–17, 1986, pp. 652–660.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[11] M. Golea and M. Marchand, "A growth algorithm for neural network decision trees," *Europhys. Lett.*, vol. 12, no. 3, pp. 205–210, 1990.

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

[13] S. Judd, *Proc. IEEE 1st Conf. Neural Networks*, vol. 2, San Diego, CA, 1987, p. 685.

[14] S. A. J. Keibek, H. M. A. Andree, M. H. F. Savenije, G. T. Barkema, and A. Taal, "A fast partitioning algorithm and a comparison of feedforward neural networks," *Europhys. Lett.*, vol. 18, pp. 555–559, 1992.

[15] J. I. Hidalgo, J. Lanchares, and J. M. Sánchez, "A method for multiple-level logic synthesis based on the simulated annealing algorithm," *Microelectron. J.*, vol. 28, pp. 143–150, 1997.

[16] M. Marchand and M. Golea, "On learning simple neural concepts: From halfspace intersections to neural decisions lists," *Network: Compu. Neural Syst.*, vol. 4, pp. 67–85, 1993.

[17] M. Marchand, M. Golea, and P. Ruján, "A convergence theorem for sequential learning in two-layer perceptrons," *Europhys. Lett.*, vol. 11, no. 6, pp. 487–492, 1990.

[18] M. Mezard and J. P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *Journal of Physics A*, vol. 22, no. 12, pp. 2191–2203, 1989.

[19] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, Expanded 1988 ed. Cambridge, MA: MIT Press, 1969.

[20] M. M. Muselli, "On sequential construction of binary neural networks," *IEEE Trans. Neural Networks*, vol. 6, pp. 678–690, 1995.

[21] J. P. Nadal, "Study of a growth algorithm for neural networks," *Int. J. Neural Syst.*, vol. 1, pp. 55–59, 1989.

[22] A. Ngom, "Synthesis of multiple-valued logic functions by neural networks," Ph.D. dissertation, Comput. Sci. Dept., Univ. Ottawa, Ottawa, ON, Canada, Oct. 1998.

[23] A. Ngom, C. Reischer, D. A. Simovici, and I. Stojmenović, "Learning with permutably homogeneous multiple-valued multiple-threshold perceptrons," *Neural Processing Lett.*, vol. 12, no. 1, Aug. 2000.

[24] A. Ngom, I. Stojmenović, and J. Žunić, "On the number of multilinear partitions and the computing capacity of multiple-valued multiple-threshold perceptrons," in *Proc. 29th IEEE Int. Symp. Multiple-Valued Logic*.

[25] Z. Obradović and I. Parberry, "Learning with discrete multivalued neurons," *J. Comput. Syst. Sci.*, vol. 49, no. 2, pp. 375–390, 1994.

[26] ——, "Computing with discrete multivalued neurons," *J. Comput. Syst. Sci.*, vol. 45, no. 3, pp. 471–492, 1992.

[27] R. L. Rivest, "Linear decision list," *Machine Learning*, vol. 2, pp. 229–246, 1987.

[28] P. Ruján and M. Marchand, "A geometric approach to learning in neural networks," *Complex Syst.*, vol. 3, pp. 229–242, 1989.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition—Volume I: Foundations*, J. L. McClelland, D. E. Rumelhart, and D. E. Rumelhart, Eds. Cambridge, MA: MIT Press, 1986.

[30] T. Sasao, "On the optimal design of multiple-valued plas," *IEEE Comput.*, vol. 38, no. 4, pp. 582–592, 1989.

[31] J. D. Schaffer, L. D. Whitley, and L. J. Eshelman, "Combinations of genetic algorithms and neural networks: A survey of the state of the art," in *COGANN-92: Int. Workshop Combinations Genetic Algorithms Neural Networks*, L. D. Whitley and J. D. Schaffer, Eds., 1992.

[32] I. K. Sethi, "Entropy nets: From decision trees to neural networks," *Proc. IEEE*, vol. 78, pp. 1605–1613, 1990.

[33] J. A. Sirat and J. P. Nadal, "Neural trees: A new tool for classification," *Network*, vol. 1, pp. 423–438, 1990.

[34] K.-Y. Siu, V. Roychowdhury, and T. Kailath, *Discrete Neural Computation: A Theoretical Foundation*, ser. Information and System Sciences. Englewood Cliffs, NJ: Prentice Hall, 1995.

[35] Z. Tang, Q. Cao, and O. Ishizuka, "A learning multiple-valued logic networks: Algebra, algorithm, and application," *IEEE Trans. Comput.*, vol. 47, pp. 247–251, 1998.

[36] Z. Tang, O. Ishizuka, and K. Tanno, "Learning multiple-valued logic networks based on back-propagation," in *Proc. 25th IEEE Int. Symp. Multiple-Valued Logic, IEEE Computer Society Technical Committee on Multiple-Valued Logic*, May 1995, pp. 270–275.

[37] G. Wang and H. Shi, "Tmlnn: Triple-valued or multiple-valued logic neural network," *IEEE Trans. Neural Networks*, vol. 9, no. 6, pp. 1099–1117, Nov. 1998.

[38] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, pp. 1423–1447, Sep. 1999.

[39] S. Young and T. Downs, "Carve: A constructive algorithm for real-valued examples," *IEEE Trans. Neural Networks*, vol. 9, no. 6, pp. 1180–1190, Nov. 1998.

[40] A. Ngom, C. Reischer, D. A. Simovici, and I. Stojmenović, "Learning with permutably homogeneous multiple-valued multiple-threshold perceptrons," in *Proc. 28th IEEE Int. Symp. Multiple-Valued Logic*, May 1998, pp. 161–166.

[41] A. Ngom, I. Stojmenović, and J. Žunić, "On the number of multilinear partitions and the computing capacity of multiple-valued multiple-threshold perceptrons," in *Proc. 25th IEEE Int. Symp. Multiple-Valued Logic, IEEE Comput. Soc. Tech. Committee on Multiple-Valued Logic*, May 1999, pp. 208–213.

**Alioune Ngom** received the B.Sc. degree in computer science from the University of Québéc at Trois-Rivières, Trois-Rivières, QC, Canada, in 1992, and the M.Sc. and Ph.D. degrees in computer science from the University of Ottawa, Ottawa, ON, Canada, in 1995 and 1998, respectively.

From August 1998 until June 2000, he was a Faculty Member at the Department of Computer Science at Lakehead University, Thunder Bay, ON, Canada. He is currently an Assistant Professor in the Computer Science Department at the University of Windsor, Windsor, ON, Canada. His current research and teaching interests include multiple-valued logic and algebra, discrete neural computation, applications of evolutionary computation, complex adaptive systems, parallel algorithms and architectures, and computational molecular biology. He currently serves as a Guest Editor for the special issue *Computational Intelligence in Multiple-Valued Logic* of the journal *Multiple-Valued Logic—An International Journal*.

**Ivan Stojmenović** received the B.S. and M.S. degrees in 1979 and 1983, respectively, from the University of Novi Sad and the Ph.D. degree in mathematics in 1985 from the University of Zagreb, Yugoslavia. He earned a third degree prize at International Mathematics Olympiad for high school students in 1976.

In 1980, he joined the Institute of Mathematics, University of Novi Sad, Yugoslavia. In fall 1988, he joined the faculty in the Computer Science Department at the University of Ottawa, ON, Canada, where he currently holds the position of a Full Professor in SITE. He has published three books and more than 140 different papers in journals and conferences. His research interests are wireless networks, parallel computing, multiple-valued logic, evolutionary computing, neural networks, combinatorial algorithms, computational geometry and graph theory. He is currently a Managing Editor of *Multiple-Valued Logic—An International Journal*, and an Editor of the following journals: *Parallel Processing Letters, IASTED International Journal of Parallel and Distributed Systems, Discrete Mathematics and Theoretical Computer Science, Parallel Algorithms and Applications, and Tangenta*.

**Veljko Milutinović** (M'81–SM'85) received the Ph.D. degree from the University of Belgrade, Serbia, Yugoslavia, in 1982.

He has been with the Department of Computer Engineering, School of Electrical Engineering, University of Belgrade since 1990. Prior to that, he was on the faculty of Purdue University, West Lafayette, IN. His research interests are in computer architecture/design, as well as in system support for electronic business on the Internet. He has contributed more than 50 papers to IEEE journals on computer architecture/design and technology-aware system support for mission-critical applications. He has consulted for leading industries in U.S.A. and Europe (IBM, RCA, NCR, AT&T, Virtual, eT, Zycad, Aerospace Corporation, Electrospace Corporation, Intel, Fairchild, Honeywell, Encore, Phillips, etc.). He was involved in a number of market successful industrial efforts (Designer of the first multiprocessor HF data modem in the 1970s, Coarchitect of the first 200MHz RISC microprocessor in the 1980s, Project Leader of the first RMS system for personal computers in the 1990s). He is the author of several books and was an Editor/Co-Editor for a number of IEEE tutorial books and conference proceedings.

Dr. Milutinović has served as a Guest Editor for special issues of PROCEEDINGS OF THE IEEE, IEEE TRANSACTIONS ON COMPUTERS, IEEE Computer, and IEEE Concurrency. He has presented more than 300 invited talks around the world.