Minimization of Multivalued Multithreshold Perceptrons Using Genetic Algorithms

Alioune Ngom, Ivan Stojmenović *   Zoran Obradović †

Abstract

We address the problem of computing and learning multivalued multithreshold perceptrons. Every n-input k-valued logic function can be implemented using a (k, s)-perceptron, for some number of thresholds s. We propose a genetic algorithm to search for an optimal (k, s)-perceptron that efficiently realizes a given multiple-valued logic function, that is to minimize the number of thresholds. Experimental results show that the genetic algorithm find optimal solutions in most cases.

1. Introduction

Let k be a fixed positive integer and let $K = \{0, \ldots, k-1\}$. A *k-valued logic function* f maps the Cartesian power K^n into K. Denote by P_k^n the set of all such functions $f : K^n \mapsto K$. The set $P_k = \bigcup_{n \geq 1} P_k^n$ is the set of all k-valued logic functions.

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. A *discrete n-input multiple-valued neuron* has a discrete transfer function and realizes a function of n variables ranging in the set $S \subseteq R$ with values in K, that is computes a function $f : S^n \mapsto K$ (where $S \subseteq R$). For $S = K$ we refer to the processing unit as a *multiple-valued logic neuron* since it simulates a multiple-valued logic function $f : K^n \mapsto K$. Multiple-valued logic neural networks are thus neural networks composed of multiple-valued logic neurons as processing units. The first model of multiple-valued logic neural networks were introduced in [2] and since then various other models have been described [9, 13, 14].

*Department of Computer Science, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario K1N 9B4, Canada, {angom,ivan}@csi.uottawa.ca. Research is partially supported by NSERC and OGS grants

†School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington 99164-2752

The problem we address in this research paper is that of learning multiple-valued logic functions by genetic algorithms. Our model of multiple-valued logic neuron is a multiple-valued multiple-threshold element. Special cases of our neuron model, where the number of thresholds is fixed to $k-1$, were introduced in literature [9, 11] and their learning power have also been investigated in [10].

Multiple-threshold devices have drawn less enthusiasm. Among there qualities, though, is that given enough thresholds, a single multiple-threshold element can realize any given function operating on a finite domain. The ability of multiple-threshold devices to simulate a larger number of functions compared to single-threshold devices is vital for the capacity and capabilities of neural networks based on threshold logic. It is therefore of practical as well as theoretical interest to develop and study learning algorithms for such neural networks.

A problem still left open in the domain of multiple-valued multiple-threshold functions is how to minimize the number of thresholds in order to construct the most efficient multiple-valued multiple-threshold networks or units. To minimize the number of thresholds, traditional techniques of multiple-valued multiple-threshold circuit synthesis use either trial-and-errors, or allow to synthesize only classes of functions for which an optimal number of thresholds can be obtained (synthesis of *k-valued symmetric functions*, for instance). The multiple-valued multiple-threshold networks considered in literature have no learning capabilities, that is, their parameters are set by the designers once and for all using some traditional techniques of networks synthesis. Also, only some small classes of k-valued logic functions are considered for multiple-valued multiple-threshold synthesis techniques. We propose genetic algorithms as minimization techniques.

2. Multiple-valued multiple-threshold perceptrons

In the theory of multiple-valued logic functions there exists a very important class of functions called *multiple-valued multiple-threshold functions* [1, 5]. Such functions are used in the design of classes of multiple-valued logic circuits called *programmable logic arrays* [12].

A *k-valued s-threshold function* of one variable [5] is defined as

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

where $\vec{o} = (o_0, \ldots, o_s) \in K^{s+1}$ is an output vector, $\vec{t} = (t_1, \ldots, t_s) \in R^s$ is a threshold vector where $t_i \le t_{i+1}$ $(1 \le i \le s-1)$, and s $(1 \le s \le k^n - 1)$ is the number of threshold values.

Let $\vec{x} = (x_1, \ldots, x_n) \in K^n$. It is well known that any n-input k-valued logic function f can be transformed into a k-valued s-threshold function $g_{k,s}^{\vec{t},\vec{o}}$ (for some s), where $y = \vec{w} \cdot \vec{x} = \sum_{i=1}^{n} w_i x_i$ is called the *excitation* and $\vec{w} = (w_1, \ldots, w_n) \in R^n$ is a weight vector associated with $\vec{x}$ [1, 5].

A *k-valued s-threshold perceptron*, abbreviated as *(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

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

where the perceptron's *transfer function* is a k-valued s-threshold function $g_{k,s}^{\vec{t},\vec{o}} : R \mapsto K$.

A (k,s)-perceptron is *monotone* if $\vec{o}$ is monotone, that is $o_0 \le \cdots \le o_s$ or $o_0 \ge \cdots \ge o_s$, otherwise it is *nonmonotone*. The multiple-valued logic neuron described in [9, 11] correspond to k-valued $(k-1)$-threshold perceptrons, that is the $(k, k-1)$-perceptrons in our definition. A (k,s)-perceptron is *homogeneous* if $\vec{o}$ is the identity permutation on K, that is $o_i = i$ for $0 \le i \le s$, otherwise it is *heterogeneous*.

A *(e,p)-permutation* (or permutation of e elements out) of $P = \{a_0, \ldots, a_{p-1}\}$ is an arrangement of e distinct elements of P, with $e \le p$. For instance, $a_1 a_2 a_4$ and $a_3 a_0 a_1$ are two distinct (3, 5)-permutations. The total number of (e,p)-permutations is $\frac{p!}{(p-e)!}$. The permutations we consider here are permutations without repetitions (i.e. without repeated elements). A (k,s)-perceptron is said to be *permutably homogeneous* if its output vector is a $(s+1, k)$-permutation. Thus for permutably homogeneous (k,s)-perceptrons we necessarily have $s \le k-1$.

2.1. Multilinear separability

The problem of computing (or simulating) a given function $f \in P_k^n$, by a (k,s)-perceptron for some s, is to determine a vector $\vec{r} = (\vec{w}, \vec{t}, \vec{o}) \in R^{n+s} \times K^{s+1}$ such that $F_{k,s}^n(\vec{r})(\vec{x}) = f(\vec{x})$ $(\forall \vec{x} \in K^n)$, i.e. $f = F_{k,s}^n(\vec{r})$. We will refer to $\vec{r}$ as a *s-representation* of $F_{k,s}^n$ for f. One interesting open question is to find *minimal* s-representation for $f \in P_k^n$. In other words, to obtain a s-representation $\vec{r}$ with the least possible number of thresholds s such that $F_{k,s}^n(\vec{r}) = f$. We will refer to this problem as the *s-representation problem* which is not equivalent to, for a *fixed* s, finding a s-representation $\vec{r}$ for f. The later problem is the focus of this paper.

In this paper, we will be mainly interested in finding *minimal* s for which there exist a s-representation for a given $f \in P_k^n$. In other words, given $f \in P_k^n$, we want to find a s-representation $\vec{r}$ with the least possible number of thresholds s such that $F_{k,s}^n(\vec{r}) = f$.

Let $V = \{\vec{x}_1, \ldots, \vec{x}_v\} \subseteq K^n$ be a set of v vectors $(v \ge 1)$. A k-valued logic function f defined over V and specified by the input-output pairs $\{(\vec{x}_1, f(\vec{x}_1)), \ldots, (\vec{x}_v, f(\vec{x}_v))\}$, where $\vec{x}_i \in K^n$, $f(\vec{x}_i) \in K$, is said to be *s-separable* if there exist vectors $\vec{w} \in R^n$, $\vec{t} \in R^s$ and $\vec{o} \in K^{s+1}$ such that

$$f(\vec{x}_i) = \begin{cases} o_0 & \text{if } \vec{w} \cdot \vec{x}_i < t_1 \\ o_j & \text{if } t_j \le \vec{w} \cdot \vec{x}_i < t_{j+1} \text{ for } 1 \le j \le s-1 \\ o_s & \text{if } t_s \le \vec{w} \cdot \vec{x}_i \end{cases} \quad (3)$$

for $1 \le i \le v$. Equivalently, f is s-separable if and only if it has a s-representation defined by $(\vec{w}, \vec{t}, \vec{o})$. A k-valued logic function defined over V is said to be *s-nonseparable* if it is not s-separable.

In other words, a (k,s)-perceptron partitions the space $V \subseteq K^n$ into $s+1$ distinct classes $H_0^{[o_0]}, \ldots, H_s^{[o_s]}$, where $H_i^{[o_i]} = \{\vec{x} \in V | f(\vec{x}) = o_i$ and $t_i \le \vec{w} \cdot \vec{x} < t_{i+1}\}$, using s parallel hyperplanes. We assume that $t_0 = -\infty$ and $t_{s+1} = +\infty$). Each hyperplane equation denoted by H_j $(1 \le j \le s)$ is of the form

$$H_j : \vec{w} \cdot \vec{x} = t_j \quad (4)$$

A function implementable by a homogeneous (k,s)-perceptron is said to be *homogeneously separable* (or homogeneous, for short). A function computable by a (k,s)-perceptron with given output vector $\vec{o}$ is said to be *$\vec{o}$-separable*. A function implementable by a (k,s)-perceptron whose output vector is monotone is said to be *monotoneously separable*. A function computable by a permutably homogeneous (k,s)-perceptron is said to be *permutably homogeneously separable* (or simply, permutably homogeneous). For instance, the function

f_1 shown in figure 1 is 3-separable, $(0,2,1,3)$-separable, nonmonotoneously separable and permutably homogeneous.

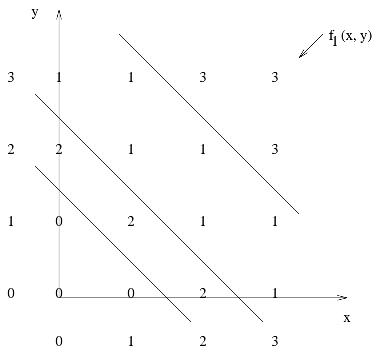

Figure 1: Three-separable two-input four-valued logic function.

Notice that if $\vec{o} \in K^{s+1}$ then every $\vec{o}$-separable function (for some $\vec{o}$) is also s-separable. However the converse is not true, that is s-separability does not imply $\vec{o}$-separability (for some $\vec{o}$). The only case where s-separability is equivalent to $\vec{o}$-separability is the two-valued one-threshold case, that is when $k = 2$, $s = 1$ and $\vec{o} = (0,1)$ or $(1,0)$. Every 1-separable two-valued logic function is $(0,1)$-separable, and also, every $(0,1)$-separable two-valued logic function is 1-separable.

2.2. The (k, s)-perceptron learning problem

For a fixed s, a threshold vector $\vec{\hat{t}}$ is *canonical* if for every k-valued logic function f, computable by a (k, s)-perceptron, there always exist vectors $\vec{w}$ and $\vec{o}$ such that $F^n_{k,s}(\vec{w}, \vec{\hat{t}}, \vec{o}) = f$. In other word, $\vec{\hat{t}}$ is canonical if every (k, s)-perceptron computable function f has a s-representation of the form $(\vec{w}, \vec{\hat{t}}, \vec{o})$, for some $\vec{w}$ and $\vec{o}$. For instance, the vector $\vec{\hat{t}} = (0)$ is canonical for a (2, 1)-perceptron and the vector $\vec{\hat{t}} = (0,1)$ is canonical for a (3, 2)-perceptron. One of the results from [11] was that there is no canonical set of thresholds for a $(k, k-1)$-perceptron when $k \geq 4$. This result which also applies to (k, s)-perceptrons in general indicates that learning algorithms which modify only the weights do not necessarily converge and that the threshold vector should be learned in addition to the weight vector.

Let $f \in P^n_k$ be a target function to learn. The (k, s)-*perceptron learning problem* is the problem of determining a s-representation for f. That is, to search for vector $\vec{r} \in R^{n+s} \times K^{s+1}$ such that $F^n_{k,s}(\vec{r}) = f$.

[10] proposed a learning algorithm for homogeneous $(k, k-1)$-perceptrons (we call it *homogeneous* $(k, k-1)$-*perceptron learning algorithm*). As a consequence of the (2, 1)-perceptron convergence theorem [6], it is proven in [10] that the homogeneous $(k, k-1)$-perceptron learning algorithm converges if and only if there exists a $(k-1)$-representation $(\vec{w}, \vec{t}, \vec{o})$ for f.

3. Computing optimal s-representations with genetic algorithms

Holland [4] first proposed *genetic algorithms* (GA) in the early 70's 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. In literature, Holland's genetic algorithm is commonly called the *Simple Genetic Algorithm* or SGA.

3.1. Problem representation

Fundamental to the GA structure is the encoding mechanism for representing the problem's variables. For the s-representation problem, the search space is the space of weight vectors $\vec{w}$ and the representation is more complex. Unfortunately, there is no practical way to encode s-representation problem as a binary chromosome to which the classical genetic operators discussed in [3] can be applied in a meaningful fashion. Therefore it is natural to represent the possible solutions as vectors $\vec{w} \in R^n$ and design appropriate genetic operators which are suitable for the s-representation problem. Each weight vector will uniquely determine a s-representation. To determine how good is a solution the GA needs a fitness function to evaluate the chromosomes.

A note on the initial population. We initialize the population with random real-coded chromosomes 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 α_i is a random number in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$. What we are trying to do in both methods of initialization is to generate random hyperplanes (since each $\vec{w}$ represent a hyperplane).

3.2. Fitness function

The objective function, the function to be optimized, provides the mechanism for evaluating each chromosome. To describe our fitness function we will need the concept of valid and invalid thresholds (hyperplanes).

To compute the thresholds for a given chromosome $\vec{w}$, we calculate for every $\vec{x} \in K^n$ the value $\vec{w} \cdot \vec{x}$ and construct a sorted array (or list) of records of the form $(\vec{w} \cdot \vec{x}, f(\vec{x}))$. The array is sorted using $\vec{w} \cdot \vec{x}$ as primary key and $f(\vec{x})$ as secondary key. Let these records be sorted as follows: $\vec{x}_1, \ldots, \vec{x}_{k^n}$, or more precisely, $(\vec{w} \cdot \vec{x}_i, f(\vec{x}_i)), 1 \le i \le k^n$, where $\vec{w} \cdot \vec{x}_i \le \cdots \le \vec{w} \cdot \vec{x}_{k^n}$. Then $\vec{w} \cdot \vec{x}_j$ is a threshold if $f(\vec{x}_{j-1}) \ne f(\vec{x}_j)$. We collect all thresholds in a list $\vec{t}$. Some thresholds in $\vec{t}$ may be duplicated (i.e. $t_{i-1} = t_i$ for some i).

Let $T(\vec{w}) = V(\vec{w}) + I(\vec{w})$, where $T(\vec{w})$ is the total number of thresholds generated by $\vec{w}$, $V(\vec{w})$ and $I(\vec{w})$ are respectively the number of valid thresholds and invalid thresholds generated by $\vec{w}$. A threshold t_i $(1 \le i \le T(\vec{w}) \le k^n - 1)$ is *valid* if all points $\vec{x} \in K^n$ lying in its corresponding hyperplane H_i (given by $\vec{w} \cdot \vec{x} = t_i$) are in the same class (i.e. $f(\vec{x})$ has the same value for all points in hyperplane H_i), otherwise it is *invalid*. In other words, invalid thresholds are those for which there exist at least two points $\vec{x}_1$ and $\vec{x}_2 \in K^n$ such that $\vec{w} \cdot \vec{x}_1 = \vec{w} \cdot \vec{x}_2$ but $f(\vec{x}_1) \ne f(\vec{x}_2)$. A hyperplane is valid (invalid) if it corresponds to valid (invalid) threshold. With these definitions then duplicated thresholds in $\vec{t}$ are invalid while non duplicated thresholds are valids.

$T(\vec{w})$ is the total number of thresholds in $\vec{t}$ and can be used to evaluate how good or bad is a chromosome. The best chromosomes are those which have the least $T(\vec{w})$. We can therefore define our fitness function as follows

$$\mathit{Fitness1}(\vec{w}) = 1 - \frac{T(\vec{w})}{k^n - 1} \tag{5}$$

Notice that a GA always maximizes an objective function and since $1 \le T(\vec{w}) \le k^n - 1$, then $\mathit{Fitness1}(\vec{w})$ is maximal when $T(\vec{w})$ is minimal.

However, invalid thresholds must need severe penalty. For instance, assume a n-input k-valued logic function $f : K^n \mapsto \{0, 1\}$ chosen at random. Then one may take hyperplanes $x_1 = 0, x_1 = 1, \ldots, x_1 = k - 1$ as invalid thresholds. These k hyperplanes (or k^2 thresholds) will separate in our sense but are not really separating as such random function needs actually an exponential number of thresholds. Because of this fact, instead of using formula 5 we can alternatively use formula 6 below.

$$\mathit{Fitness2}(\vec{w}) = \frac{2 - \frac{T(\vec{w})}{k^n-1} - \frac{I(\vec{w})}{T(\vec{w})}}{2} = 1 - \frac{T(\vec{w})}{2 \cdot (k^n - 1)} - \frac{I(\vec{w})}{2 \cdot T(\vec{w})} \tag{6}$$

Here we not only minimize $T(\vec{w})$ (in second term) but we also punish a chromosome that generates a large number of invalid hyperplanes (in last term). That is we are minimizing $T(\vec{w})$ and $I(\vec{w})$ at the same time. Notice that $0 \le I(\vec{w}) \le T(\vec{w})$ and thus $\mathit{Fitness2}(\vec{w})$ will be maximal if both $T(\vec{w})$ and $I(\vec{w})$ are minimal.

In all our experiments, both formulae of fitness yield the same results for $I(\vec{w}) = 0$. We do not know for now how they do behave for $I(\vec{w}) \ne 0$ since the $\vec{w}$'s generated valid thresholds only. The probability to generate invalid thresholds seems to be very close to zero.

A note on the time complexity of the evaluation function. For a given $\vec{w}$, it takes $n \cdot k^n$ steps to compute all the $\vec{w} \cdot \vec{x}$'s, $k^n \cdot \log k^n$ steps to sort them and at most k^n steps to compute $T(\vec{w})$. Therefore the evaluation of $\mathit{Fitness}(\vec{w})$ has a time complexity of $O(n \cdot k^n \cdot \log k)$.

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

3.3. Crossover

Crossover is the GA's crucial operation. Pairs of randomly selected chromosomes are subjected to crossover. For the s-representation problem we propose the following mixed crossover method for real-coded chromosomes. 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

$$\vec{c}_1 = \vec{p}_1 + \vec{p}_2 \tag{7}$$

and

$$c_{2_i} = \begin{cases} p_{1_i} & \text{if } random() \leq 0.5 \\ p_{2_i} & \text{otherwise} \end{cases} \quad (8)$$

Child $\vec{c}_1$ is simply the addition of its parents and is assured to be their exact middle vector since the parents are unit vectors. Child $\vec{c}_2$ is a uniform crossover of its parents, that is, at coordinate i each parent have 50% chances to be selected as c_{2_i} $(1 \leq i \leq n)$. Crossover is applied only if a randomly generated number in the range 0 to 1 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.

3.4. Mutation

After crossover, chromosomes are subjected to random mutations. We propose two methods of *coordinate-wise* mutations. Both methods are similar to the bitwise mutation (for binary chromosomes). Let $\vec{p}$ be a *unit* vector to be mutated to a child $\vec{c}$.

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

$$c_i = random(-1, 1) \quad (9)$$

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

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

$$c_i = rand(-1, 1) \cdot \sqrt{1 - p_i^2} \quad (10)$$

where $rand(-1, 1)$ returns -1 or 1 with equal probability (random sign).

Just as p_{cros} controls the probability of crossover, the mutation rate p_{muta} gives the probability for a given coordinate to be mutated.

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 $p_m \leq 0.1$.

4. Experiments and discussions

In our experiments, the control parameters' setting for the GA were: population size $p = 100$; number of generations $g = 1000$; crossover probability $p_{cros} = 0.75$; and mutation probability $p_{muta} = 0.005$. The most important parameters here are p, p_{cros} and p_{muta} and the values used for them seem to be optimal in that they yield better results in all experiments we have done. The high crossover rate is necessary to widen the search while the low mutation rate is necessary to avoid too much chromosome disruptions. Because we use an *elitist strategy* some best chromosome in a current generation is always reproduced to the next generation in order to avoid lost of good genetic material. We use a large population size to preserve the diversity of the population, that is to avoid premature convergence. The fact that we used a mixed crossover technique also helps maintain the diversity. In all experiments, we used *Fitness2* as our evaluation function. Also we used *stochastic universal selection shceme* as our reproduction method.

It is interesting that the proposed population representation does not depend on k. It make us wonder how the number of invalid thresholds vary with k (or n). For a fixed n (or k), larger k (or n) means smaller separation among classes and these problems are typically more difficult to learn. We did some experiments on random functions with small k versus large k and small n versus large n in order to see how the number of invalid thresholds changes. The number of invalid thresholds obtained (in all experiments) is always zero. Although our approach is slow, it is slower as n grows than as k grows.

We tested our GA on random permutably homogeneous functions of the form

$$f(\vec{x}) = \lfloor (\sum_{i=1}^{n} \frac{1}{a_i} x_i) + n \rfloor \bmod k \quad (11)$$

where $a_i = 2i+1$, $2 \leq k \leq 4$ and $2 \leq n \leq 7$. Each of these functions defines itself its separating hyperplanes and their number. The number of hyperplanes is simply the number of distinct values of such function minus one, and each hyperplane H_j is defined by the equation $\sum_{i=1}^{n} \frac{1}{a_i} x_i = t_j$ for some threshold t_j $(1 \leq t \leq$ number of thresholds). The output vector can also be obtained by computing the value of f for $x_1 = \cdots = x_n = 0$ and $x_1 = \cdots = x_n = k - 1 = 3$ and listing in increasing order modulo k the sequence of other distinct values of f in between.

The difficulty for the GA to find an optimal solution within 1000 generations depends mostly on n rather than k. This is not surprising since the search space

is exponential on n and thus the GA needs more and more generations to successfully obtain an optimum. For $k \geq 4$ and $n \geq 5$, for example, the GA could not find an optimum within ten runs of 1000 generations each, however it was successfull within one run with 2000 generations. This suggest that given enough time the GA will always find the minimal s-representation for a logic function.

It is interesting to note that the permutably homogeneous functions are the most difficult for the GA since their s-representations are very small. This indicates that for most (random) functions the GA will perform much better than for permutably homogeneous functions because s is larger on average.

We compared our technique with the *extended permutably homogeneous* (k, s)*-perceptron learning algorithm* (EPHPLA) described in [8]. It is proven in [8] that the EPHPLA always converges for permutably homogeneous functions, and that also, it always finds a minimal s-representation for such functions. The EPHPLA is faster and outperform the GA on learning these functions within one run of 1000 learning epochs. The GA converged better only for $n = 2$. The main advantage of the GA method over the EPHPLA is that it can learn any logic function provided enough time is given.

5. Conclusion

We have used genetic algorithms to minimize multivalued multithreshold perceptrons for computing given functions. Experiments show that the genetic search can be very effective however slow it may be. Generalization properties of the GA can be studied by modifying the fitness function to work with proper subsets of K^n.

References

[1] M.H. Abd-El-Barr, S.G. Zaky and Z.G. Vranesic (1986), *Synthesis of multivalued multithreshold functions for CCD implementation*, IEEE Transactions on Computers, V.C-35, N.2, February, pp.124-133.

[2] S.C. Chan, L.S. Hsu and H.H. Teh (1988), *On neural logic networks*, Neural Networks, Pergamon Press, vol.1, supplement I, p.428.

[3] D.E. Goldberg (1989), *Genetic algorithms in search, optimization, and machine learning*, Reading, MA, Addison-Wesley.

[4] J.H. Holland (1975), *Adaptation in natural and artificial systems*, Ann Arbor, MI, Michigan University Press.

[5] O. Ishizuka (1976), *Multivalued multithreshold networks*, Proceedings of the 6th IEEE International Symposium on Multiple-Valued Logic, pp.44-47.

[6] M. Minsky and S. Papert (1969), *Perceptrons: An introduction to computational geometry*, Cambridge, MA: MIT Press, Expanded edition 1988.

[7] A. Ngom (1998), *Synthesis of multiple-valued logic functions by neural network*, Ph.D. Thesis, Computer Science Department, University of Ottawa, Ottawa, in progress.

[8] A. Ngom, C. Reischer, D.A. Simovici and I. Stojmenović (1998), *Learning with permutably homogeneous multiple-valued multiple-threshold perceptrons*, Proceedings of the 28th IEEE International Symposium on Multiple-Valued Logic, in this issue.

[9] Z. Obradović (1996), *Computing with nonmonotone multivalued neurons*, Multiple-Valued Logic - An International Journal, V.1, N.4, pp.271-284.

[10] Z. Obradović and I. Parberry (1994), *Learning with discrete multivalued neurons*, Journal of Computer and System Sciences, V.49, N.2, pp.375-390.

[11] Z. Obradović and I. Parberry (1992), *Computing with discrete multivalued neurons*, Journal of Computer and System Sciences, V.45, N.3, pp.471-492.

[12] T. Sasao (1989), *On the optimal design of multiple-valued PLAs*, IEEE Computer, V.C-38, N.4, pp.582-592.

[13] Z. Tang, O. Ishizuka and K. Tanno (1995), *Learning multiple-valued logic networks based on backpropagation*, Proceedings of the 25th IEEE International Symposium on Multiple-Valued Logic, pp.270-275.

[14] T. Watanabe, M. Matsumoto, M. Enokida and T. Hasegawa (1990), *A design of multi-valued logic neuron*, Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic, pp.418-425.