*Neural Processing Letters* **12:** 71–90, 2000.
© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*


# Learning with Permutably Homogeneous Multiple-Valued Multiple-Threshold Perceptrons

ALIOUNE NGOM[1], CORINA REISCHER[2], DAN A. SIMOVICI[3] and IVAN STOJMENOVIĆ[4]

[1] *Computer Science Department, Lakehead University, 955 Oliver Road, Thunder Bay, Ontario P7B 5E1, Canada; E-mail: angom@ice.lakeheadu.ca*
[2] *Department of Mathematics and Computer Science, University of Quebec at Trois-Rivieres, Trois-Rivieres, Quebec G9A 5H7, Canada; E-mail: corina_reischer@uqtr.uquebec.ca*
[3] *Department of Mathematics and Computer Science, University of Massachusetts at Boston, Boston, Massachusetts 02125, USA; E-mail:dsim@umb.edu*
[4] *Department of Computer Science, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario K1N 9B4, Canada*

**Abstract.** The $(n, k, s)$-perceptrons partition the input space $V \subset R^n$ into $s+1$ regions using $s$ parallel hyperplanes. Their learning abilities are examined in this research paper. The previously studied homogeneous $(n, k, k-1)$-perceptron learning algorithm is generalized to the permutably homogeneous $(n, k, s)$-perceptron learning algorithm with guaranteed convergence property. We also introduce a high capacity learning method that learns any permutably homogeneously separable $k$-valued function given as input.

**Key words:** learning, multiple-valued multiple-threshold functions, multilinear separability, partial order set, perceptrons.

## 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 $K^n$ (of all ordered $n$-tuples of elements of $K$) 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 $P_k = \bigcup_{n \geqslant 1} P_k^n$ 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 [2, 11, 21, 23].

The problem we address in this research paper is that of learning multiple-valued logic functions by multiple-valued logic neurons. Our model of multiple-valued logic neuron is a multiple-valued multiple-threshold element with learning capability. Special cases of our neuron model, where the number of thresholds is fixed to $k-1$, were introduced in literature [11, 13, 15–17] and their learning power have also been investigated in [12, 14]. Chan *et al.* [2] developed a model of three-valued logic neuron that simulate *Kleenean functions* (these are special class of three-valued logic functions) using the well known back-propagation learning algorithm as learning method.

## 2. Multiple-Valued Multiple-Threshold Perceptrons

In this section we present a more formal definition of multiple-valued multiple-threshold neurons and address their learning problem. Other definitions associated with our model will also be given.

In the theory of multiple-valued logic functions there exists a very important class of functions called *multiple-valued multiple-threshold functions* [1, 4–6]. Such functions are used in the design of classes of multiple-valued logic circuits called *programmable logic arrays* [18, 19].

A *n-input k-valued s-threshold function* of one variable [6] is defined as

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

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 $t_i \leqslant t_{i+1}$ $(1 \leqslant i \leqslant s-1)$, and $s$ $(1 \leqslant s \leqslant k^n - 1)$ 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 $g_s^{\vec{t},\vec{o}}$ (for some $s$), where $y = \vec{w}\vec{x} = \sum_{i=1}^n w_i x_i$ 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, 6].

A *n-input k-valued s-threshold perceptron*, 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

$$F_{k,s}^n(\vec{w}, \vec{t}, \vec{o})(\vec{x}) = g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x}) = \begin{cases} o_0 & \text{if } \vec{w}\vec{x} < t_1 \\ o_i & \text{if } t_i \leqslant \vec{w}\vec{x} < t_{i+1} \; 1 \leqslant i \leqslant s-1 \\ o_s & \text{if } t_s \leqslant \vec{w}\vec{x} \end{cases} \tag{2}$$

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

A $(n, k, s)$-perceptron is *monotone* if $\vec{o}$ is monotone, that is $o_0 \leqslant \cdots \leqslant o_s$ or $o_0 \geqslant \cdots \geqslant o_s$, otherwise it is *nonmonotone.* The multiple-valued logic neuron described in [11, 13] correspond to $k$-valued $(k-1)$-threshold perceptrons, that is the $(n, k, k-1)$-perceptrons in our definition. Also, the three-valued logic neurons introduced in [2] are (n,3,2)-perceptrons with threshold vector $\vec{t} = (-d, +d)$, $d \in R$, and output vector $\vec{o} = (1, 0, 2)$. A $(n, k, s)$-perceptron is *homogeneous* if $\vec{o}$ is the identity permutation on $K$, that is $o_i = i$ for $0 \leqslant i \leqslant s$, otherwise it is *heterogeneous.*

A $p$-*permutation* (or permutation of $p$ elements out) of $P = \{a_0, \ldots, a_{p-1}\}$ is an arrangement of $p > 0$ elements into $p$ positions. For example, $a_0a_3a_2a_4a_1$ and $a_3a_0a_1a_4a_2$ are two different five-permutations. The order of the elements is important. There are $p!$ distinct $p$-permutations. A $(e, p)$-*permutation* (or permutation of $e$ elements out) of $P$ is an arrangement of $e$ distinct elements of $P$, with $e \leqslant p$. For instance, $a_1a_2a_4$ and $a_3a_0a_1$ are two distinct (3, 5)-permutations. The total number of $(e, p)$-permutations is $\frac{p!}{(p-e)!}$. When $e = p$ we obtain $p$-permutations. The permutations we consider here are permutations without repetitions (i.e. without repeated elements). A $(n, k, s)$-perceptron is said to be *permutably homogeneous* if its output vector is a $(s+1, k)$-permutation. Thus for permutably homogeneous $(n, k, s)$-perceptrons we necessarily have $s \leqslant k-1$.

### 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.

Let $V = \{\vec{x}_1, \ldots, \vec{x}_v\} \subseteq K^n$ be a set of $v$ vectors $(v \geqslant 1)$. 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

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

for $1 \leqslant i \leqslant v$. 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 $(n, k, s)$-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 $t_i \leqslant \vec{w}\vec{x} < t_{i+1}\}$, using $s$ parallel hyperplanes. We assume that $t_0 = -\infty$ and $t_{s+1} = +\infty$). Each hyperplane equation denoted by $H_j$ $(1 \leqslant j \leqslant s)$ is of the form

$$H_j : \vec{w}\vec{x} = t_j \tag{4}$$

A function implementable by a homogeneous $(n, k, s)$-perceptron is said to be *homogeneously separable* (or homogeneous, for short). A function computable by a $(n, k, s)$-perceptron with given output vector $\vec{o}$ is said to be $\vec{o}$-*separable.* A function implementable by a $(n, k, s)$-perceptron whose output vector is monotone is said to be *monotoneously separable.* A function computable by a permutably homogeneous $(n, k, s)$-perceptron is said to be *permutably homogeneously* separable (or simply, permutably homogeneous). For instance, the functions $f_1$, $f_2$ and $f_3$ shown in Figure 1 are all 3-separable and, moreover, are respectively $(0, 2, 1, 3)$-separable, $(0, 1, 2, 3)$-separable and $(3, 2, 1, 3)$-separable; $f_1$ is nonmonotoneously separable and permutably homogeneous; $f_2$ is (permutably) homogeneous and monotoneoulsy separable; $f_3$ is non permutably homogeneous and nonmonotoneously separable.

Notice that since $\vec{o} \in K^{s+1}$ 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 implies $\vec{o}$-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 $(n, k, s)$-PERCEPTRON LEARNING PROBLEM

For a fixed $s$, a threshold vector $\vec{t}$ 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 $F_{k,s}^n(\vec{w}, \vec{t}, \vec{o}) = f$. In other word, $\vec{t}$ is canonical if every computable function $f$ has

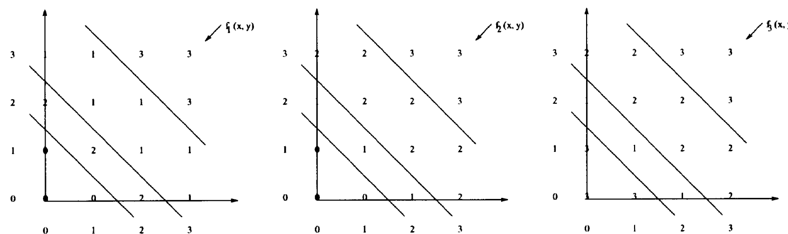

*Figure 1.* Examples of three-separable two-input four-valued logic functions.

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 (n,2, 1)-perceptron and the vector $\vec{\hat{t}} = (0, 1)$ is canonical for a (n,3, 2)-perceptron. One of the results from [13] was that there is no canonical set of thresholds for a $(n, k, k-1)$-perceptron when $k \geqslant 4$. This result which also applies to $(n, k, s)$-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 $f \in P_k^n$ 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 $F_{k,s}^n(\vec{r}) = f$.

Obradović and Parberry [12, 14] proposed a learning algorithm for homogeneous $(n, k, k-1)$-perceptrons (we call it *homogeneous $(n, k, k-1)$-perceptron learning algorithm*). As a consequence of the (n,2, 1)-perceptron convergence theorem [3, 7, 8, 10], it is proven in [12] that the homogeneous $(n, k, k-1)$-perceptron learning algorithm converges if and only if there exists a $(k-1)$-representation $(\vec{w}, \vec{t}, \vec{o})$ for $f$.

The homogeneous $(n, k, k-1)$-perceptron learning algorithm can only learn functions in the class of homogeneous $k$-valued logic functions. This is the main limitation of the algorithm since its learnable class of functions is a tiny portion of the set of $s$-separable $k$-valued logic functions (for a fixed $s$). In the next section we propose a $(n, k, s)$-perceptron learning algorithm that learn a larger class of functions.

For a fixed number of thresholds $s$, the problem of learning a $(k, s)$-perceptron, in general, still remains open. A more difficult learning problem is the case where the vector $\vec{o}$ is unknown and that therefore it should be found along with the vectors $\vec{w}$ and $\vec{t}$. For instance, $f_1$ cannot be computed (hence cannot be learned) by a homogeneous (4, 3)-perceptron. This function can be learned (hence computed) by a permutably homogeneous (4, 3)-perceptron with $g_{4,3}^{\vec{t},\vec{o}=(0,2,1,3)}$ as nonmonotone transfer function. A monotone $(k, s)$-perceptron cannot simulate a nonmonotoneously separable $k$-valued logic function.

## 3. Permutably Homogeneous $(n, k, s)$-Perceptrons

In this section we propose a learning algorithm for permutably homogeneous $(k, s)$-perceptrons. A $(n, k, s)$-perceptron is *permutably homogeneous* if its output vector is a $(s+1, k)$-permutation. To the best of our knowledge there are no known learning algorithms for $(n, k, s)$-perceptrons in general. Our algorithms, when applied to the special case $s = k - 1$, that is the $(n, k, k-1)$-perceptrons, are more powerful than the homogeneous $(n, k, k-1)$-perceptron learning algorithm described in [12, 14] in that they can learn a larger class of multiple-valued logic functions.

### 3.1. PERMUTABLY HOMOGENEOUS $(n, k, s)$-PERCEPTRON LEARNING ALGORITHM

A permutation does not need to contain all values of $K$. There are $k$-valued logic functions whose set of output values is a subset $S \subseteq K$, that is functions of the form $f : K^n \mapsto S \subseteq K$. Examples of such functions are, for instance, functions of the form $f : K^n \mapsto \{0, 1\}$, that is functions with $k$-valued inputs and two-valued outputs.

Let $\vec{o} \in K^{s+1}$ be the output vector of a $(n, k, s)$-perceptron. When $\vec{o}$ is a $(s+1, k)$-permutation, that is there are no $i$ and $j$ $(i \neq j)$ such that $o_i = o_j$, we propose the *permutably homogeneous* $(n, k, s)$*-perceptron learning algorithm* (for a fixed $(s+1, k)$-permutation $\vec{o}$) as shown in Figure 2.

In Figure 2, the constant $0 < \eta \leqslant 1$ is the learning rate. The initial weights can be set to any (random) values. The initial thresholds can also be set to any (random) values, however, empirical tests show that the algorithm converges faster when the initial thresholds are set in such a way that $t_{i+1} - t_i = c$ (e.g. $c = kn$) and that $t_{v-1} \leqslant t_v \leqslant t_{v+1}$ each time we update $t_v$ or $t_{v+1}$. We can also generate a new random $\eta$ before each call to *MultiPerceptronUpdate*. $Pos_{\vec{o}}[z]$ is the position (or the index) of $z$ in $\vec{o}$. For example, if $\vec{o} = (3, 0, 2, 1)$ then $Pos_{\vec{o}}[3] = 0$, $Pos_{\vec{o}}[0] = 1$, $Pos_{\vec{o}}[2] = 2$ and $Pos_{\vec{o}}[1] = 3$.

In the algorithm, the weight and threshold vectors are always updated in opposite directions using the error value $\delta = Pos_{\vec{o}}[f(\vec{x})] - Pos_{\vec{o}}[v]$. If $Pos_{\vec{o}}[f(\vec{x})] < Pos_{\vec{o}}[v]$ then

```
Procedure MultiPerceptron(f, n, k, s, o⃗);
    w⃗ := 0⃗;
    t⃗ := 0⃗;
    Repeat
        for each x⃗ ∈ K^n do
            v := F^n_{k,s}(w⃗, t⃗, o⃗)(x⃗);
            if f(x⃗) ≠ v then
                MultiPerceptronUpdate(x⃗, f(x⃗), v, o⃗);
            Output (w⃗, t⃗);
    Until F^n_{k,s}(w⃗, t⃗, o⃗) = f;

Procedure MultiPerceptronUpdate(x⃗, f(x⃗), v, o⃗);
    δ := Pos_o⃗[f(x⃗)] − Pos_o⃗[v];
    if δ < 0 then
        t_{Pos_o⃗[v]} := t_{Pos_o⃗[v]} − ηδ;
    if δ > 0 then
        t_{Pos_o⃗[v]+1} := t_{Pos_o⃗[v]+1} − ηδ;
    for 1 ≤ i ≤ n do
        w_i := w_i + ηδx_i;
```

*Figure 2.* Permutably homogenous $(n, k, s)$-perceptron learning algorithm.

$\delta < 0$ means that the weights are too large or $t_{Pos_{\vec{o}}[v]}$ is too small. Therefore we decrease the weights and increase $t_{Pos_{\vec{o}}[v]}$. If $Pos_{\vec{o}}[f(\vec{x})] > Pos_{\vec{o}}[v]$ then $\delta > 0$ means that the weights are too small or $t_{Pos_{\vec{o}}[v]+1}$ is too large. Thus we increase the weights and decrease $t_{Pos_{\vec{o}}[v]+1}$. When $Pos_{\vec{o}}[f(\vec{x})] = Pos_{\vec{o}}[v]$ no modification is done and the algorithm goes to the next step. So, $\vec{w}$ and $\vec{t}$ are always updated in opposite directions given by the position of $f(\vec{x})$ relative to that of $v$ in $\vec{o}$. Notice that $\vec{o}$ is known and given as input to the algorithm.

Our algorithm can learn any $\vec{o}$-separable logic function as long as $\vec{o}$ is a fixed $(s+1, k)$-permutation. In other words, the algorithm can learn any function whose input vectors can be separated by a set of parallel hyperplanes (i.e. $s$-separable, for some $s$) and whose classes — separated by these hyperplanes — have distinct values (i.e. $\vec{o}$-separable, for some $(s+1, k)$-permutation $\vec{o}$). In fact, the permutably homogeneous algorithm generalizes the homogeneous algorithm in that $\vec{o}$ is any permutation (such permutation does not need to be the identity permutation nor a $k$-permutation).

When $\vec{o}$ is not a permutation, that is there are $i$ and $j$ ($i \neq j$) such that $o_i = o_j$, then it becomes difficult to obtain a learning algorithm with guaranteed convergence. This problem is left open for further research.

### 3.2. PERMUTABLY HOMOGENEOUS $(n, k, s)$-PERCEPTRONS CONVERGENCE PROPERTIES

Given a $(s+1, k)$-permutation $\vec{o}$, experiments show that the permutably homogeneous $(k, s)$-perceptron learning algorithm always converges for $\vec{o}$-separable functions. That is, given the appropriate output vector any permutably homogeneous $k$-valued logic function will be learned. In this section we give a formal proof of convergence of the algorithm given in Figure 2.

The *latency* (or *delay*) of a learning algorithm is the worst case running time between the output of one set of assignments and the next. We will assume *unit-cost* latency; that is we will assume that the algorithm is implemented on a digital computer with word-size large enough that each elementary arithmetic and logic operation can be implemented in constant time. The mistake bound is the worst case total number of distinct assignments outputs. The latency and the mistake bound of the permutably homogeneous $(n, k, s)$-perceptron learning algorithm are, respectively, $O(n)$ and $\Omega(k^n)$.

Let $\vec{\pi} = (\pi(0), \ldots, \pi(k-1))$ be a $k$-permutation. The identity permutation is denoted by $\vec{\sigma}$. Thus a homogeneous $(n, k, s)$-perceptron is a neuron whose output vector is $\vec{\sigma}$ and whose $s = k - 1$. It has been proven in [12, 14] that the homogeneous $(n, k, k-1)$-perceptron learning algorithm always terminates in learning homogeneous $k$-valued logic functions.

Let $f \in P_k^n$ be a homogeneous function, then we will denote by $f_\pi \in P_k^n$ the function obtained by permuting the output values of $f$ with respect to $\pi$. That is, we define the homogeneous transformation $f_\pi(\vec{x}) = \pi(f(\vec{x}))$ for all $\vec{x} \in K^n$. Clearly $f_\sigma = f$. The function $f_\pi$ is a permutably homogeneous function. In fact, the set of all permutably

homogeneous $(k-1)$-separable functions can be constructed in this way from the set of all homogeneous functions.

LEMMA 3.1. *Let $f$ be homogeneous and $\vec{\pi}$ be a k-permutation. Then $\vec{r}_\sigma = (\vec{w}, \vec{t}, \vec{\sigma})$ is a $(k-1)$-representation for $f$ if and only if $\vec{r}_\pi = (\vec{w}, \vec{t}, \vec{\pi})$ is a $(k-1)$-representation for $f_\pi$.*

*Proof.* $\Rightarrow$) If $\vec{r}_\sigma$ is a $(k-1)$-representation for $f$ then, for $\vec{x} \in K^n$, we have $f_\pi(\vec{x}) = \pi(f(\vec{x})) = \pi(F^n_{k,k-1}(\vec{r}_\sigma)(\vec{x})) = \pi(g^{\vec{t},\vec{\sigma}}_{k,k-1}(\vec{w}\vec{x})) = \pi(\sigma(i)) = \pi(i) = g^{\vec{t},\vec{\pi}}_{k,k-1}(\vec{w}\vec{x}) = F^n_{k,k-1}(\vec{r}_\pi)(\vec{x})$ (if $t_i \leqslant \vec{w}\vec{x} \leqslant t_{i+1}$ with $t_0 = -\infty$ and $t_k = +\infty$). So $\vec{r}_\pi$ is a $(k-1)$-representation for $f_\pi$. $\Leftarrow$) Similar proof using the fact that, for $z \in K$, we have $\sigma(z) = \pi^{-1}(\pi(z)) = z$ where the permutation $\pi^{-1}$ is the inverse of $\pi$ ($\pi^{-1}$ is guaranteed to exist and is unique since the set of all permutations on $K$ forms a group). □

Lemma 3.1. tells us that, given a $(k-1)$-representation $(\vec{w}, \vec{t}, \vec{\sigma})$ for a homogeneous function $f$, the homogeneous transformation of $f$ into a permutably homogeneous function $f_\pi$ leaves the weights and the thresholds invariant. Therefore, the positions of the $k-1$ separating parallel hyperplanes do not change after transformation of $f$. Clearly, in Figure 1 the three hyperplanes remain invariant even after transformation of $f_2$ into $f_1 = f_{2_{(0,2,1,3)}}$ and vice versa.

THEOREM 3.1. *Given the output vector $\vec{\pi}$, the permutably homogeneous $(n, k, k-1)$-perceptron learning algorithm for learning a function $f \in P^n_k$ terminates if and only if $f$ is $\vec{\pi}$-separable.*

*Proof.* $\Rightarrow$) If the permutably homogeneous $(n, k, k-1)$-perceptron algorithm with output vector $\vec{\pi}$ terminates on learning $f$ then a $(k-1)$-representation $(\vec{w}, \vec{t}, \vec{\pi})$ exists for $f$. Therefore $f$ is $\vec{\pi}$-separable and thus permutably homogeneous. $\Leftarrow$) Let $f$ be $\vec{\pi}$-separable, we want to show that the algorithm terminates for $f$. The algorithm, instead of learning $f$, learns $f_{\pi^{-1}}$ using the homogeneous $(n, k, k-1)$-perceptron learning algorithm with output vector $\vec{\pi}^{-1}$. Since $f$ is permutably homogeneous and $\vec{\pi}$-separable then from the $\Leftarrow$ part of Lemma 3.1 we have that $f_{\pi^{-1}}$ is homogeneous and thus $\vec{\sigma}$-separable. Once $f_{\pi^{-1}}$ has been learned, $f$ can be reconstructed using the $\Rightarrow$ part of Lemma 3.1. The algorithm is guaranteed to terminate by the homogeneous $(n, k, k-1)$-perceptron convergence theorem of [12]. □

If in Figure 2 we replace $\vec{o}$ by $\vec{\pi}$, then we clearly have that $Pos_{\vec{o}} = \pi^{-1}$ and therefore the algorithm truly learns $f_{\pi^{-1}}$ and reconstruct $f$ using the parameters found for $f_{\pi^{-1}}$.

The results above concern output vectors which are $k$-permutations of $K$, i.e. full permutations. Next, we consider the case of $(s+1, k)$-permutations for fixed $s \leqslant k-1$. In the sequel we let $\vec{v} = (v(0), \ldots, v(s))$ be a $(s+1, k)$-permutation and $\vec{v}^c = (v^c(0) = v(0), \ldots, v^c(s) = v(s), v^c(s+1), \ldots, v^c(k-1))$ be the $k$-permutation (full permutation) obtained by adding $k-s$ coordinates to $\vec{v}$.

LEMMA 3.2. *$f$ is $\vec{v}$-separable if and only if it is $\vec{v}^c$-separable.*

*Proof.* Clearly, if $\vec{r} = (\vec{w}, \vec{t} = (t_1, \ldots, t_s), \vec{v})$ is a $s$-representation for $f$, then it is easy to see that $\vec{r}^c = (\vec{w}, \vec{t}^c = (t_1^c = t_1, \ldots, t_s^c = t_s, t_{s+1}^c, \ldots, t_{k-1}^c), \vec{v}^c)$ is a $(k-1)$-representation for $f$ and vice versa (where $t_{s+1}^c \leqslant \cdots \leqslant t_{k-1}^c$ are arbitrary and their corresponding hyperplanes contain no points between them). □

THEOREM 3.2. (Permutably homogeneous perceptron convergence theorem) *Given the output vector $\vec{v}$, the permutably homogeneous $(n, k, s)$-perceptron learning algorithm for learning a function $f \in P_k^n$ terminates if and only if $f$ is $\vec{v}$-separable.*

*Proof.* $\Rightarrow$) Same as in Theorem 3.1. $\Leftarrow$) The algorithm learns $f$ using the output vector $\vec{v}^c$ instead of $\vec{v}$. Since from Lemma 3.2 $f$ is $\vec{v}^c$-separable, then by Theorem 3.1 the algorithm is guaranteed to terminate. □

### 3.3. DETERMINING THE OUTPUT VECTORS OF PERMUTABLY HOMOGENEOUS $(n, k, s)$-PERCEPTRONS

A more difficult learning problem is when the output vector $\vec{o}$ is not known and that it should be determined along with $\vec{w}$ and $\vec{t}$. In this section we let $\vec{o}$ be a $(s+1, k)$-permutation. An obvious solution to this problem is to generate each $(s+1, k)$-permutation $\vec{p}$ and apply the learning algorithm with $\vec{o} = \vec{p}$. This method takes $O(enk^n \frac{k!}{(k-s-1)!})$ time complexity (where $e$ is the number of learning epochs) and thus is non realistic for even small values of $k$. A better method is, for a given function $f \in P_k^n$, to search for a partial order relation defined over $f$'s values and, if such order relation exists then we search for a good linear extension of the corresponding partially ordered set and use it as the output vector of a $(n, k, s)$-perceptron. Before we describe the algorithm we need a few definitions.

A *partially ordered set* $(P, <_P)$ or (*poset*) is a set $P = \{a_0, \ldots, a_{p-1}\}$ equipped with an irreflexive, antisymmetric and transitive relation $<_P$. A *linear extension* $L$ of $P$ is a linear (or total) ordering $<_L$ of the elements of $P$ such that $a_i <_L a_j$ whenever $a_i <_P a_j$. For example, $L = \{1 <_L 0 <_L 3 <_L 2 <_L 4\}$ and $L = \{4 <_L 1 <_L 3 <_L 0$

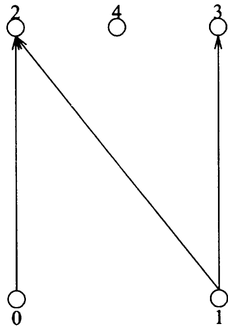


*Figure 3.* Example of partially ordered set.

$<_L 2\}$ are two linear extensions of the poset shown in Figure 3. We also define a *covering relation* $\prec_P$ between two elements of $P$: $x \prec_P y$ if and only if $x <_P y$ and there is no $z$ such that $x <_P z <_P y$.

Our extended permutably homogeneous $(n, k, s)$-perceptron learning algorithm, for searching an output vector $\vec{o} \in K^{s+1}$ and then learning a given function $f$ using $\vec{o}$, is shown in Figure 4. For $f \in P_k^n$, let $K_f$ be its set of values. Denote by $<_{K_f}^d$ an order relation over $K_f$ with respect to the $d$-th variable, that is $x_d$, where $1 \leqslant d \leqslant n$. We will refer to $d$ as *direction* since, as we will see later, it selects the dimension of the $n$-cube $K^n$ along which we construct a poset.

The *extended learning algorithm* goes as follows. Using the *partial order construction algorithm* of Figure 5 we attempt to construct a poset $(K_f, <_{K_f}^d)$ with respect to some variable $x_d$. If such $(K_f, <_{K_f}^d)$ exists and is a chain then $\vec{o}$ is the concatenation of the unique linear extension of $(K_f, <_{K_f}^d)$ and a $(s - |K_f| + 1, |K - K_f|)$-permutation of $K - K_f$ and it will be used to learn $f$. If such $(K_f, <_{K_f}^d)$ exists but is not a chain then we attempt to obtain $(K_f, <_{K_f}^{d+1})$ and so on until either we construct a chain poset $(K_f, <_{K_f}^d)$ for some $d$, or there is some $d$ such that a poset $(K_f, <_{K_f}^d)$ cannot be obtained, or $d = n$. When $n$ non-chain posets $(K_f, <_{K_f}^1), \ldots, (K_f, <_{K_f}^n)$ are constructed then, using the *partial orders combination algorithm* of Figure 6, we attempt to combine these $n$ non-chain posets into a chain poset $(K_f, <_{K_f})$.

The *partial order construction algorithm* goes as follows. For a given direction $1 \leqslant d \leqslant n$, we start with an antichain $(K_f, <_{K_f}^d)$. Then, for every $\vec{x} = (x_1, \ldots, x_n)$, we construct poset $(K_f, <_{K_f}^d)$ by adding new comparable pairs $f_1 = f(x_1, \ldots, x_d, \ldots, \ldots, x_n) <_{K_f}^d f(x_1, \ldots, x_d + 1, \ldots, x_n) = f_2$ whenever $f_1 \nless_{K_f}^d f_2$, and also, we add new comparable pairs $y <_{K_f}^d f_2$ whenever $y <_{K_f}^d f_1$ and comparable pairs

```
Procedure ExtendedLearning(f, n, k, s);
    K_f ⊆ K := set of output values of f;
    Contradiction := False;
    Unique := False;
    d := 0;
    Repeat
        d := d + 1;
        ConstructPartialOrder(Contradiction, (K_f, <^d_{K_f}), Unique);
        If Unique = True Then
            e⃗ := (e_0, ..., e_{|K_f|-1}) the unique linear extension of (K_f, <^d_{K_f});
            p⃗ := (p_0, ..., p_{s-|K_f|}) a (s - |K_f| + 1, |K - K_f|)-permutation of K - K_f;
            o⃗ := (e_0, ..., e_{|K_f|-1}, p_0, ..., p_{s-|K_f|});
            MultiPerceptron(f, n, k, s, o⃗);
    Until Contradiction = True or Unique = True or d = n;
    If Contradiction = False and Unique = False then
        CombinePartialOrders((K_f, <^1_{K_f}), ..., (K_f, <^n_{K_f}));
```

*Figure 4.* Extended $(n, k, s)$-perception learning algorithm.

```
Procedure ConstructPartialOrder(Contradiction, (K_f, <^d_{K_f}), Unique);
   (K_f, <^d_{K_f}) := antichain;
   While 0 ≤ x_1 ≤ k − 1 and Contradiction = False do
      ⋱
         While 0 ≤ x_d ≤ k − 2 and Contradiction = False do
            ⋱
               While 0 ≤ x_n ≤ k − 1 and Contradiction = False do
                  f_1 := f(x_1, ..., x_d, ..., x_n);
                  f_2 := f(x_1, ..., x_d + 1, ..., x_n);
                  If f_1 ≮^d_{K_f} f_2 then
                     If f_2 <^d_{K_f} f_1 then
                        Contradiction := True;
                     Else
                        Add comparable pair f_1 <^d_{K_f} f_2 in (K_f, <^d_{K_f});
                        For every value y such that y <^d_{K_f} f_1 do
                           If f_2 <^d_{K_f} y then
                              Contradiction := True;
                              Exit the for loop;
                           Else
                              Add comparable pair y <^d_{K_f} f_2 in (K_f, <^d_{K_f});
                        For every value y such that f_2 <^d_{K_f} y do
                           If y <^d_{K_f} f_1 then
                              Contradiction := True;
                              Exit the for loop;
                           Else
                              Add comparable pair f_1 <^d_{K_f} y in (K_f, <^d_{K_f});
   If Contradiction = False and (K_f, <^d_{K_f}) is a chain then
      Unique := True;
```

*Figure 5.* Partial order construction algorithm.

$f_1 <^d_{K_f} y$ whenever $f_2 <^d_{K_f} y$, for some $y$. We exit the loops as soon as there is some new comparable pair $y <^d_{K_f} z$ (for some $y$ and $z$) that cannot be added to $(K_f, <^d_{K_f})$. That is $z <^d_{K_f} y$ is already in $(K_f, <^d_{K_f})$ and, therefore, adding its inverse leads to inconsistency. In this case, the construction of $(K_f, <^d_{K_f})$ cannot be completed along the direction $d$ (meaning that $(K_f, <^d_{K_f})$ simply do not exist). In case $(K_f, <^d_{K_f})$ exists — its construction can be completed along $d$ — then it has a unique linear extension if and only if it is a chain. In other words, $(K_f, <^d_{K_f})$ is always constructed in the positive direction along the $d$-th dimension of the $n$-cube $K^n$, or equivalently, starting from any point $\vec{y}$ in the hyperplane $x_d = 0$ we move toward the hyperplane $x_d = k - 1$ by following the line segment orthogonal to both hyperplanes and whose

Procedure CombinePartialOrders$((K_f, <^1_{K_f}), \ldots, (K_f, <^n_{K_f}))$;
    *Contradiction* := False;
    *Unique* := False;
    $(K_f, <_{K_f}) := (K_f, <^1_{K_f})$;
    $d := 1$;
    $c_d := 0$;
    Repeat
        If *Contradiction* = True then
            CutPoset(*Contradiction*, $(K_f, <_{K_f})$, $[c_1 \ldots c_d]$, *Unique*);
        Else
            If $d < n$ then
                ExtendPoset(*Contradiction*, $(K_f, <_{K_f})$, $[c_1 \ldots c_d]$, *Unique*);
    Until *Unique* = True or $c_1 = 1$ or (*Contradiction* = False and $d = n$);
    If $c_1 = 1$ or (*Contradiction* = False and $d = n$) then
        *Unique* := True; {we force unicity if we did not find a consistent chain}
        $(K_f, <_{K_f}) := (K_f, <^1_{K_f})$; {or any poset above with the smallest width}
    If *Unique* = True then
        Repeat
            $\vec{e} := (e_0, \ldots, e_{|K_f|-1})$ a linear extension of $(K_f, <_{K_f})$;
            $\vec{p} := (p_0, \ldots, p_{s-|K_f|})$ a $(s - |K_f| + 1, |K - K_f|)$-permutation of $K - K_f$;
            $\vec{o} := (e_0, \ldots, e_{|K_f|-1}, p_0, \ldots, p_{s-|K_f|})$;
        Until MultiPerceptron$(f, n, k, s, \vec{o})$ terminates;

*Figure 6.* Partial orders combination algorithm.

Procedure ExtendPoset(*Contradiction*, $(K_f, <_{K_f})$, $[c_1 \ldots c_d]$, *Unique*);
    $d := d + 1$;
    $c_d := 0$;
    $(K_f, <_{K_f}) := (K_f, <_{K_f}) \oplus (K_f, <^d_{K_f})$;
    {Check for contradiction and unicity during $\oplus$ operation};

*Figure 7.* Extension algorithm.

Procedure CutPoset(*Contradiction*, $(K_f, <_{K_f})$, $[c_1 \ldots c_d]$, *Unique*);
    While $c_d = 1$ do
        $(K_f, <_{K_f}) := (K_f, <_{K_f}) \ominus (K_f, >^d_{K_f})$;
        $d := d - 1$;
    $c_d := 1$;
    $(K_f, <_{K_f}) := (K_f, <_{K_f}) \ominus (K_f, <^d_{K_f})$;
    $(K_f, <_{K_f}) := (K_f, <_{K_f}) \oplus (K_f, >^d_{K_f})$;
    {Check for contradiction and unicity during $\oplus$ operation};

*Figure 8.* Cutting algorithm.

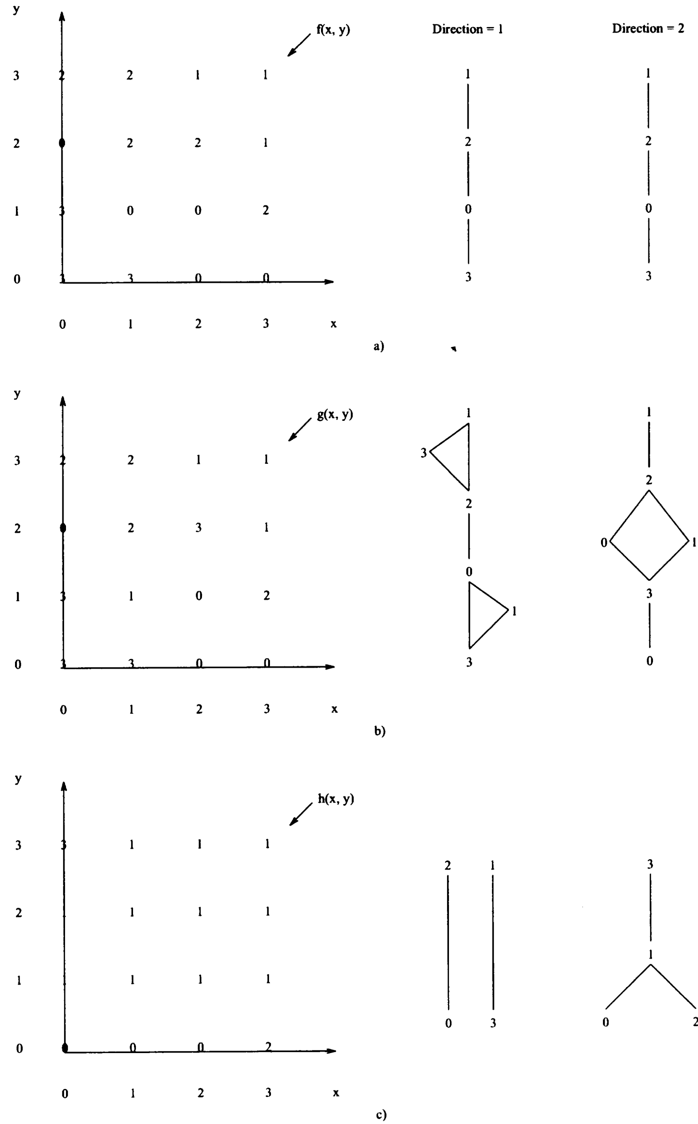


*Figure 9.* Constructed partial orders for some $f \in P_4^2$.

origin is $\vec{y}$. Figure 9 shows examples of constructed posets. For illustration purpose, we have also shown the graphs obtained from function $g(x, y)$ (Figure 9b) when attempting to complete the construction with possible contradictory pairs. As one can see, such graph cannot be embedded into a poset. So posets $(K_g, <^1_{K_g})$ and $(K_g, <^2_{K_g})$ do not exist.

For given permutably homogeneous and $s$-separable function $f \in P_k^n$, not any linear extension $\vec{e}$ of a non-chain $(K_f, <^d_{K_f})$ is good for learning. The $(n, k, s)$-perceptron

learning algorithm may not terminate when $\vec{e}$ is used. For instance, from Figure 9c the linear extension $(3, 0, 1, 2)$ of $(K_h, <^1_{K_h})$ is not good for learning $h$ since $h$ is $(2, 0, 1, 3)$-separable (assuming $s = 3$). Some directions may give more information on order than others. For example, $f(x_1, x_2) = (x_1 + 1) \bmod k$ is irrelevant on $x_2$ (or direction 2) and so $(K_f, <^2_{K_f})$ is an antichain whereas $(K_f, <^1_{K_f})$ is a chain. In general, given a direction $d$ there are two possibilities for failure. Either $(K_f, <^d_{K_f})$ cannot be constructed, or $(K_f, <^d_{K_f})$ exists but is a non-chain poset such that a selected linear extension (among its many linear extensions) does not yield a convergence of the $(n, k, s)$-perceptron learning algorithm (but some other will do so). Because of this fact, when a non-chain poset $(K_f, <^d_{K_f})$ is obtained for any direction $1 \leqslant d \leqslant n$, we must combine these $n$ posets in some way in order to obtain a unique linear extension. Next we describe how to combine them.

The *partial orders combination algorithm* goes as follows. Let $(K_f, <_{K_f})$ be a combination poset of $d$ consistent non-chains posets $(K_f, <^1_{K_f})$, ..., $(K_f, <^d_{K_f})$. Initially $(K_f, <_{K_f})$ is set to $(K_f, <^1_{K_f})$ and is constructed according to some binary string $c = [c_1 \dots c_d] \in \{0, 1\}^d$, where $1 \leqslant d \leqslant n$. When $c_i = 1$ then the inverse of $(K_f, <^i_{K_f})$, that is poset $(K_f, >^i_{K_f})$, is in $(K_f, <_{K_f})$, otherwise $(K_f, <^i_{K_f})$ itself is in $(K_f, <_{K_f})$ (obviously, $(K_f, <^i_{K_f})$ and $(K_f, >^i_{K_f})$ cannot both be in $(K_f, <_{K_f})$ at the same time). The combination poset $(K_f, <_{K_f})$ is constructed using an algorithm for generating binary strings of lengths $\leqslant n$ in lexicographic order. For example, for $n = 4$, the lexicographic generation of binary strings of lengths $\leqslant 4$ goes in the following manner:

| | | | |
|---|---|---|---|
| 0 | 00 | 000 | 0000 |
| | | | 0001 |
| | | 001 | 0010 |
| | | | 0011 |
| | 01 | 010 | 0100 |
| | | | 0101 |
| | | 011 | 0110 |
| | | | 0111 |
| 1 | ... | | |
| ⋮ | | | |

The algorithm is in *ExtendPoset* phase when it goes from left to right staying in a row. It is in *CutPoset* phase when the algorithm shifts to some row (possibly far) below. The algorithm is used to construct poset $(K_f, <_{K_f})$ as follows. If with the string $[c_1 \dots c_d]$ poset $(K_f, <_{K_f})$ exists but is a non-chain for $d < n$ then we extend to next string $[c_1 \dots c_d c_{d+1} = 0]$ in the row and add poset $(K_f, <^{d+1}_{K_f})$ into $(K_f, <_{K_f})$. If with the string $[c_1 \dots c_d]$ poset $(K_f, <_{K_f})$ cannot be constructed then we do not need to extend since poset $(K_f, <_{K_f})$ simply do not exist and that we cannot add a poset to an undefined poset. Hence, in this case we can bypass the lexicographic generation of binary strings to an appropriate point: we say that the algorithm is in *CutPoset* phase. We cut in the following manner. Starting from position $d$ of $c$

we search for the first position $r \leqslant d$ such that $c_r = 0$. We remove posets $(K_f, <^r_{K_f})$ and $(K_f, >^{r<i \leqslant d}_{K_f})$ from $(K_f, <_{K_f})$ and add poset $(K_f, >^r_{K_f})$ into $(K_f, <_{K_f})$, then finally, we set $d$ to $r$ and $c_d$ to 1.

To summarize, with a given string $[c_1 \dots c_d]$ we have three possibilities. We generate the next string whenever $d < n$ and $(K_f, <_{K_f})$ is a non-chain poset (the extension phase) and update $(K_f, <_{K_f})$ accordingly. We bypass the lexicographic generation to some row below whenever $(K_f, <_{K_f})$ cannot be constructed (the cutting phase) and compute $(K_f, <_{K_f})$ appropriately. We exit the algorithm as soon as $(K_f, <_{K_f})$ is a chain or $c_1 = 1$ or, $(K_f, <_{K_f})$ is a non-chain and $d = n$. In the first case we learn $f$ using the unique linear extension of $(K_f, <_{K_f})$. In the second case, when no chain poset $(K_f, <_{K_f})$ is found, we may either randomly select one poset $(K_f, <^i_{K_f})$ and look for a good linear extension of it to learn $f$, or we select among all posets constructed so far the one which has the smallest width (since it will have the smallest number of linear extensions to search); the selection can be done by computing the width of the currently constructed consistent poset and keeping track of the smallest width (and storing the associated poset). The *width* of a poset is the size of its longest antichain.

Also we do not need to continue generating new binary strings when $c_1$ becomes 1. Because they are symmetric to (i.e. complement of) those generated already (the poset constructed according to a string $c$ is dual to the poset constructed according to the complement of $c$, and hence both posets behave exactly the same way). See Figure 10 for examples of combination posets. Next we explain how to add into or remove from a combination poset $(K_f, <_{K_f})$.

Given $(K_f, <^d_{K_f})$ to be added to $(K_f, <_{K_f})$ the addition $(K_f, <_{K_f}) \oplus (K_f, <^d_{K_f})$ is defined by $(K_f, <_{K_f}) \oplus (K_f, <^d_{K_f}) = (K_f, <_{K_f} \cup \gamma(<^d_{K_f}))$, where $\gamma(<^d_{K_f})$ is the transitive closure of every comparable pair of relation $<^d_{K_f}$ in relation $<_{K_f}$. That is $O(|K_f|^2)$ comparabilities from $(K_f, <^d_{K_f})$ are added to $(K_f, <_{K_f})$ during addition. Also, for every such comparability from $(K_f, <^d_{K_f})$ its transitive closure in $(K_f, <_{K_f})$ is also added, that is $O(|K_f|)$ more comparabilities. In sum, operation $\oplus$ take $O(|K_f|^3)$ steps.

Given a poset $(K_f, <^d_{K_f})$ to be removed from poset $(K_f, <_{K_f})$, the substraction $(K_f, <_{K_f}) \ominus (K_f, <^d_{K_f})$ is defined by $(K_f, <_{K_f}) \ominus (K_f, <^d_{K_f}) = \bigcup_{[c_1 \dots c_i \dots c_{d-1}]} (K_f, <^i_{K_f})$ (where $(K_f, <^i_{K_f})$ is reversed when necessary). That is, to remove $(K_f, <^d_{K_f})$ from $(K_f, <_{K_f})$ is equivalent to restore $(K_f, <_{K_f})$ in the state it was before $(K_f, <^d_{K_f})$ was added to it. To achieve efficiency, substraction operation is done in the following way. Whenever we add a poset $(K_f, <^d_{K_f})$ in $(K_f, <_{K_f})$ we store into a separate data structure $R_d$ all comparabilities of $(K_f, <^d_{K_f})$ that are not in $(K_f, <_{K_f})$. So that when we later remove $(K_f, <^d_{K_f})$ from $(K_f, <_{K_f})$ we will eliminate only comparabilities of $R_d$ from $(K_f, <_{K_f})$. The $\oplus$ operation modified in this way still operates in $O(|K_f|^3)$. $O(|K_f|^2)$ comparabilities of $R_d$ are removed from $(K_f, <_{K_f})$. Therefore substraction takes $O(|K_f|^2)$ steps hence faster than addition. Inconsistency and unicity can be tested, respectively, in $O(1)$ and $O(|K_f|)$ during addition.

In Figure 9 we show examples of constructed posets $(K_f, <^1_{K_f})$ and $(K_f, <^2_{K_f})$ for some $f \in P^2_4$. Suppose $s = 3$. Poset $(K_f, <^1_{K_f})$ and $(K_f, <^2_{K_f})$ are chains (Figure 9a),

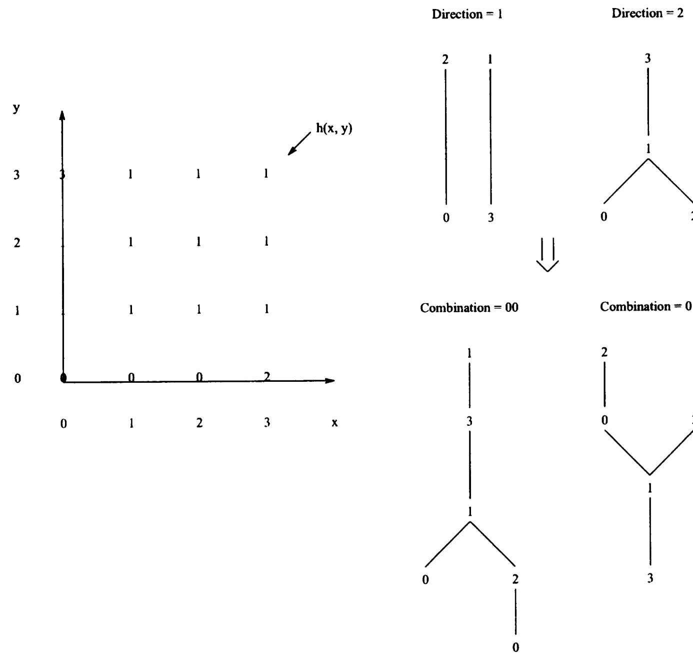


*Figure 10.* Examples of combinations posets for Figure 9c.

so they have unique linear extensions and thus $f$ is permutably homogeneous and 3-separable and can be learned. In an attempt to construct $(K_g, <^1_{K_g})$ and $(K_g, <^2_{K_g})$ we obtain graphs (Figure 9b) that cannot be embedded into posets because of inconsistencies, so $g$ is not permutably homogeneous and 3-separable and thus $g$ cannot be learned. Poset $(K_h, <^1_{K_h})$ and $(K_h, <^2_{K_h})$ are both non-chains (Figure 9c), so they have many linear extensions and $h$ is permutably homogeneous and 3-separable; however we do not know which linear extensions are good for learning $f$, so we must combine $(K_h, <^1_{K_h})$ and $(K_h, <^2_{K_h})$ to search for a unique linear extension. In Figure 10 we show two combination posets $(K_h, <_{K_h})$ according to binary strings 00 and 01. As we can see, with string 00 poset $(K_h, <_{K_h})$ cannot be obtained because of inconsistencies whereas with string 01 it is a chain.

A *thick s-separable function* is a function $f \in P^n_k$ for which the distance between any two neighboring separating hyperplanes, in any direction, is strictly greater than one.

THEOREM 3.3. *If a permutably homogeneous function $f \in P^n_k$ is thick s-separable then $(K_f, <^d_{K_f})$ is a chain for any $1 \leqslant d \leqslant n$.*

*Proof.* Let $a$ and $b$ be two neighboring distinct values connected by an edge. There is at least one separating hyperplane between them. However, there is at most one separating hyperplane since otherwise two such separating hyperplanes will be at distance strictly less than one along the dimension of that edge. Thus $a \prec^d_{K_f} b$ or

$b \prec^d_{K_f} a$, that is $a$ and $b$ are neighbors in the poset. All such neighboring pairs are detected in at least one dimension. Therefore $(K_f, <^d_{K_f})$ has a unique linear extension. □

For some non-thick $s$-separable functions, all combination posets $(K_f, <_{K_f})$ may have many linear extensions and the last *repeat* loop of the algorithm in Figure 6 can be modified as follows to make it more efficient. In parallel using several processors, we generate each linear extension of $(K_f, <^d_{K_f})$ and test it for learning, until one processor succeeds. This can be simulated on one processor by time sharing, that is, generate linear extensions and test each of them for the same time in succession, until one successfully terminates. Next we discuss the time complexity of the extended learning algorithm.

The worst case scenario, in terms of time complexity, for the partial order construction algorithm is when there is no contradiction for a given direction $d$. So the *while* loop associated with the selected variable $x_d$ will be iterated $k-1$ times and the $n-1$ remaining *for* loops associated with non-selected variables will be iterated each $k$ times. Also, each of the two inner *for* loops will be iterated $|K_f|$ times and it takes $O(|K_f|)$ steps to test whether $(K_f, <^d_{K_f})$ is a chain. Therefore the partial order construction algorithm has a time complexity of $O(k^n|K_f|)$.

The worst case scenario for the partial orders combination algorithm is when there is no contradiction for $d < n$ but always contradiction for $d = n$. So $2^n - 2$ combination posets are constructed each either by extension or by cutting and then $O(|K_f|!)$ linear extensions are checked for learning $f$. Extension and cutting involve $\oplus$ operations and tests for unicity and inconsistency. Cutting is slower than extension since it also involves $\ominus$ operations and a search for the first bit equals to 0 (starting from the end of the current string). Therefore extension and cutting take respectively $O(|K_f|^3)$ and $O(n|K_f|^2)$ steps. The $(n, k, s)$-perceptron learning algorithm takes $O(enk^n)$ steps ($e$ is the number of learning epochs) and thus the partial orders combination algorithm has $O(2^n n|K_f|^2 + (s + enk^n)|K_f|!)$ time complexity. Since in practice $e$ is large and that $2^n \leqslant k^n$ and $|K_f|^2 \leqslant |K_f|!$ then the complexity becomes $O(enk^n|K_f|!)$.

The worst case scenario for the extended learning algorithm is when poset $(K_f, <^d_{K_f})$ is a non-chain for any direction $d$. So $n$ posets are constructed and combined. Consequently, the extended learning algorithm has $O(nk^n|K_f| + enk^n|K_f|!)$, that is $O(enk^n|K_f|!)$ time complexity.

Recall the first method: generate each $(s+1, k)$-permutation $\vec{p}$ and apply the $(k, s)$-perceptron learning algorithm with output vector $\vec{o} = \vec{p}$ for learning $f$ until the learning terminates for some permutation $\vec{p}$. This method takes $O(enk^n \frac{k!}{(k-s-1)!})$ time complexity. Let us refer to it as the *permutation generating learning algorithm*. Next we compare the extended algorithm to the permutation algorithm.

First, note that the time complexity of the permutation generating algorithm is always the same for any function. That is not true for the extended algorithm. For instance, for any non permutably homogeneous function $f \in P^n_k$ the extended

algorithm takes $O(nk^n|K_f|)$ steps; the algorithm takes $O(enk^n)$ steps for any permutably homogeneous thick $s$-separable function. The worst time complexity is achieved only for permutably homogeneous non-thick $s$-separable functions $f$ whose any combination poset $(K_f, <_{K_f})$ is a non-chain or cannot be constructed. We believe that the probability to obtain such function $f$ is very close to zero (if not equal to zero), so that in practice, the extended learning algorithm runs in $O(enk^n)$ for permutably homogeneous $s$-separable functions. This proves its superiority over the permutation generating learning algorithm.

## 4. Experiments

We tested our extended learning algorithm on non-permutably homogeneous functions and on permutably homogeneous thick or non-thick functions. We could not obtain, however, non-thick functions whose combination posets are all non-chains. This suggests that such function are very rare if not inexistent. The non-thick functions we used have at least one chain combination poset. In our test we set the learning rate $\eta$ to 0.5 and the maximum number of learning epochs $e$ to 5000. We experimented with different values of $n$ and $k$. Also, the number of threshold $s$ was not given to the learning algorithm, it was to be found by the algorithm itself. The initial weigth vector is set to $\vec{0}$ and the initial threshold vector is set to $(k^n, 2k^n, \ldots, sk^n)$ after $s$ was found.

For non-permutably homogeneous functions, the algorithm behaved as expected, that is no learning is effected on these functions. For permutably homogeneous (thick or non-thick) functions the algorithm always terminated after learning the function with its unique linear extensions.

Next we discuss an example of non-thick function which we have used in our experiment for $k = 4$ and $n = 3$. Consider the two-place function $h$ shown in Figure 10. To obtain a three-place function $f$ we project the values $h$ (which will correspond to points in plane $x_3 = 0$) in the three planes $x_3 = 1, x_3 = 2, x_3 = 3$. So, $f$ is also a non-thick function as $h$. Now, to make it more difficult to find one of its *good* linear extensions we replace the value 3 that lies in plane $x_3 = 0$ by value 1, and also change the value 2 that lies in plane $x_3 = 3$ to 0. Here the function $f$ has no unique linear extension at all in any direction and thus the extended learning algorithm must combine the three constructed posets to search for a good linear extension. The algorithm did indeed, as we expected, find a chain poset which has the unique linear extension (2, 0, 1, 3). The function has been learned successfully in 61 learning epochs. We also obtain same results when extending $f$ to a $(n \geqslant 3)$-place functions.

Examples of permutably homogeneous thick functions are given by the following formula:

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

where $a_i = 2i + 1$. For example, we tested with the 4-place 4-valued logic function $f(\vec{x}) = \lfloor \frac{x_1}{3} + \frac{x_2}{5} + \frac{x_3}{7} + \frac{x_4}{9} + 4 \rfloor \bmod 4$. Clearly, such function is permutably homogeneous since it defines itself its separating hyperplanes and their number. It is easy to see that the function has three possible values, namely 0, 1 and 2 and thus there must be 2 separating hyperplanes, also, the three classes of input are separated in the order $(0, 1, 2)$. So we expect that our extended learning algorithm will find 2 separating hyperplanes and the output vector $(0, 1, 2)$. Indeed, the function was learned successfully in 946 learning epochs after the algorithm has found the output vector.

## 5. Conclusion

In this paper we have discussed an extended learning algorithm for learning the class of permutably homogeneous multiple-valued logic functions. The algorithm extends previous results from literature and achieves a better capacity than those. When the number of thresholds is not fixed, the algorithm will always find the minimal one to be used for learning a separable function. For further research we are working on the case of learning a function when the output vector is a $(s + 1, k)$-permutation with possible repetitions of its distinct elements. Another interesting research problem is to develop learning algorithm for multilayer neural networks whose processing units are $(n, k, s)$-perceptrons. Such network would have the ability to learn any $k$-valued logic function.

## References


1. Abd-El-Barr, M. H., Zaky, S. G. and Vranesic, Z. G.: (1986), Synthesis of multivalued multithreshold functions for CCD implementation, *IEEE Transactions on Computing,* **C-35**(2) (1986), 124–133.
2. Chan, S. C., Hsu, L. S. and Teh, H. H.: (1988), On neural logic networks, *Neural Networks*, Pergamon Press, 1998, p. 428.
3. Duda, R. O. and Hart, P. E.: *Pattern classification and scene analysis*, Wiley, New York, 1973.
4. Haring, D. (1965), Multithreshold threshold elements, *IEEE Transactions on Electronic and Computer*, **EC-15** (1965), 45–65.
5. Ishizuka, O.: On MVMT networks composed of conventional threshold elements, *IEEE Transactions on Computing*, **C-26**(2) (1977), 1251–1257.
6. Ishizuka, O.: Multivalued multithreshold networks, *Proceedings of the 6th IEEE International Symposium on Multiple-Valued Logic*, pp. 44–47.
7. Minsky, M. and Papert, S. *Perceptrons: An Introduction to Computational Geometry*, Cambridge, MA: MIT Press, Expanded edition, 1988.
8. Nilsson, N. J.: *Learning Machines*, McGraw-Hill, New York, 1962.
9. Ngom, A.: Synthesis of multiple-valued logic functions by neural network, Ph.D. Thesis, Computer Science Department, University of Ottawa, Ottawa, 1988.
10. Novikoff, A.: On convergence proofs for perceptrons, *Proceedings of the Symposium on Mathematical Theory of Automata*, New York, pp. 615–622.

11. Obradović, Z.: Computing with nonmonotone multivalued neurons, *Multiple-Valued Logic – An International Journal*, **1**(4) (1996), 271–284.
12. Obradović, Z. and Parberry, I.: Learning with discrete multivalued neurons, *Journal of Computer and System Sciences*, **49** (1994), 375–390.
13. Obradović, Z. and Parberry, I.: Computing with discrete multivalued neurons, *Journal of Computer and System Sciences*, **45**(3) (1992), 471–492.
14. Obradović, Z. and Parberry, I.: Learning with discrete multivalued neurons, *Proceedings of the Seventh International Conference on Machine Learning*, Austin, TX, pp. 392–399.
15. Obradović, Z. and Parberry, I.: Analog neural networks of limited precision I: Computing with multilinear threshold functions, In: D. S. Touretzky (ed.) *Advances in Neural Information Processing Systems II*, Morgan-Kaufmann, San Mateo, CA, pp. 702–709.
16. Obradović, Z. and Yan, P.: Small depth polynomial size neural networks, *Neural Computation*, **2** (1990), 402–404.
17. Obradović, Z. and Yan, P.: Lower bounds for limited precision analog neural networks, *Technical Report* CS-90-28, Department of Computer Sciences, Pennsylvania State University.
18. Sasao, T.: On the optimal design of multiple-valued PLAs, IEEE Computer, **C-38**(4) (1989), 582–592.
19. Sasao, T.: Multiple-valued logic and optimization of programmable logic arrays, *IEEE Computer*, (1988), 71–80.
20. Siu, K.-Y., Roychowdhury, V. and Kailath, T.: Discrete neural computation: A theoretical foundation, Prentice Hall Information and System Sciences Series, Thomas Kailath, Series editor, pp. 68–103.
21. Tang, Z., Ishizuka, O. and Tanno, K.: Learning multiple-valued logic networks based on back-propagation, *Proceedings of the 25th IEEE International Symposium on Multiple-Valued Logic*, pp. 270–275.
22. Trotter, W. T.: *Combinatorics and Partially Ordered Sets – Dimension Theory*, John Hopkins University Press, pp. 215–217.
23. Watanabe, T., Matsumoto, M., Enokida, M. and Hasegawa, T.: A design of multi-valued logic neuron, *Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic*, pp. 418–425.