INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

SYNTHESIS OF MULTIPLE-VALUED LOGIC FUNCTIONS BY NEURAL NETWORKS

By

Alioune Ngom

Thesis

Submited to the School of Graduate Studies and Research
In Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy
In
Computer Science

Under the auspices of the Ottawa-Carleton Institute for
Computer Science

School of Information Technology and Engineering

University of Ottawa
Ottawa, Ontario, Canada

Fall 1998.

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

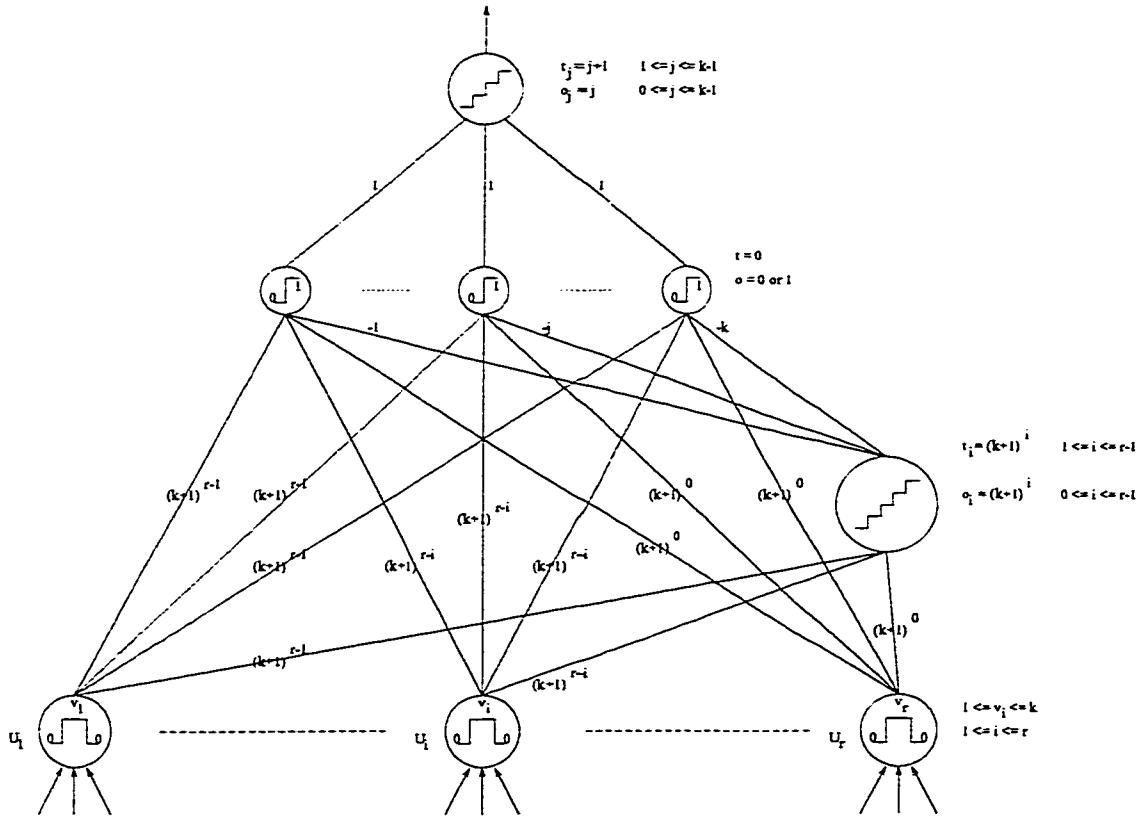
Our file Notre référence

0-612-36787-8

Canada

Synthesis of Multiple-Valued Logic Functions by Neural Networks

By Alioune Ngom



Abstract

The issue we address in this thesis is that of implementing multiple-valued logic systems by neural networks. In particular we discuss models of multiple-valued logic neurons (and neural networks) and how such models might be used to learn or compute multiple-valued logic functions. 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 in literature that analog neurons of limited precision are essentially discrete multiple-valued neurons. Our research focuses on the synthesis of multiple-valued functions by some models of multiple-valued neural networks. We introduce an interesting model of multiple-valued neuron, the *multiple-valued multiple-threshold perceptron*, which extends and generalizes the well-known binary perceptron and other previously studied models. Beside investigating the computational and learning abilities of this main model, we discuss some methods of optimizing identified resources of multiple-valued neural networks. Our main contributions in the field of neural networks and multiple-valued logic are summarized as follows. We obtain bounds on the computing capacity of multiple-valued multiple-threshold perceptrons. We obtain a high capacity learning algorithm for the multiple-valued multiple-threshold perceptron with guaranteed convergence properties. We show two techniques of constructing minimal multiple-valued neural networks for given but arbitrary functions. We propose a method for minimizing the number of thresholds of multiple-valued perceptrons. We introduce a neuro-genetic approach to minimization of multiple-valued logic expressions.

Contents

List of Figures

List of Tables

Synthesis of Multiple-Valued Logic Functions by Neural Networks

Alioune Ngom

November 24, 1998

Acknowledgment

I wish to express my deep gratitude to Dr Ivan Stojmenović, my supervisor, for his guidance and support. He has shown me by its example the importance of patience and organization in research, and taught me how to write mathematics correctly. This work is more his creation than mine. I am most appreciative of his valuable advises and his patient assistance.

I would like to thank Dr Corina Reischer from whom I have learned the importance of perseverance and good discipline in research since my early undergraduate years at the University of Québec at Trois-Riviéres. She initiated my research career and, by her help and assistance, made this work possible. Since the undertaking of my thesis, she has been a constant source of moral support and guidance.

I am very grateful to Dr Robert Holte, Dr Stan Matwin, Dr Robert Laganiére, Dr Evangelos Kranakis and Dr Dan Albert Simovici, who all accepted to be members of my Ph.D research committee or Ph.D examination commitee. Particular thanks to Dr Simovici, my external examiner, who gave me valuable advises on how to prepare and give a good oral presentation in theoretical computer science.

I wish to thank my brothers and sisters and parents for their encouragement. This would have been a daunting task if they had not been there for me.

Finally, many thanks to National Science and Engineering Research Council of Canada, to Ontario Graduates Scholarships, to School of Graduate Studies and Research (University of Ottawa), for their financial support during my studies at the University of Ottawa. Many thanks also to friends and colleagues and staffs at the Computer Science Department of the University of Ottawa.

Thanks to God for all

Chapter 1

Introduction

1.1 Artificial neural networks

Several novel methods of computation have recently emerged that are collectively known as soft computing. The raison d'être of these modes is to exploit the tolerance for imprecision and uncertainty in real-world problems to achieve tractability, robustness and low cost. Soft computing is usually used to find an approximate solution to a precisely or imprecisely formulated problem. Neural computing, fuzzy computing and evolutionary computing are the major components of this approach.

Artificial neural networks are an attempt to mimic some or all of the characteristics of biological neural networks. This soft computational paradigm differs from a programmed instruction sequence in that information is stored in form of weights. Each neuron is an elementary processor with primitive operations, like summing the weighted inputs coming to it and then amplifying or thresholding the sum. Assembly of such neurons can, in principle, perform universal computations for suitably chosen weights.

A well known model of neuron studied extensively in [Minsky 69] is called the *perceptron* (see Figure 1.1). The perceptron computes a weighted sum of its input signals and generates an output of 1 if this sum is above a certain threshold $t \in R$. Otherwise, an output of 0 results. Generally speaking, given a weight vector $\vec{w} = (w_1, \ldots, w_n) \in R^n$ and an input vector $\vec{x} = (x_1, \ldots, x_n) \in R^n$, such neuron computes a simple function of the form $f_{\vec{w}} : R^n \mapsto S$, where $n \geq 1$, $S \subseteq R$ and

$$f_{\vec{w}}(\vec{x}) = g(\vec{w}\vec{x}) \tag{1.1}$$

for some *transfer function* $g : R \mapsto S$. There are two choices for the

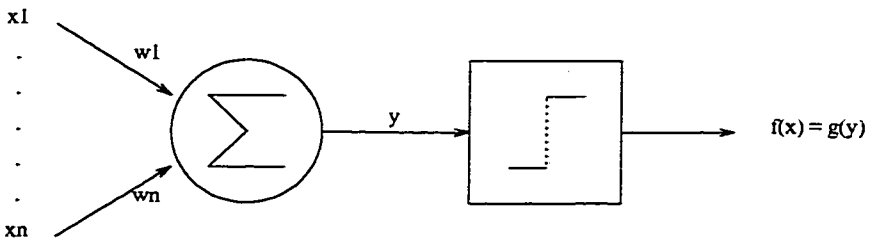


Figure 1.1: Perceptron.

set S currently popular in literature. The first is the *discrete model* with $S = \{0, 1\}$. In this case, if a neuron's threshold is t, then its transfer function g is a *linear threshold function* (see Figure 1.2a) defined by

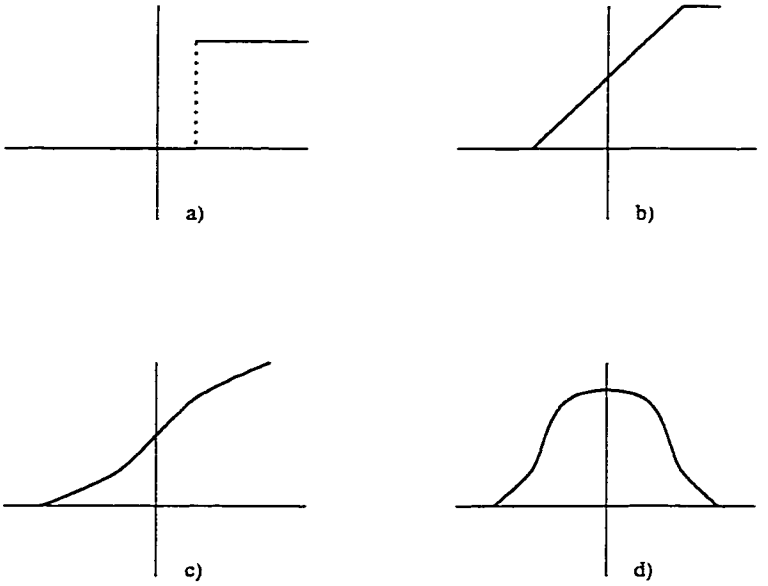


Figure 1.2: Examples of transfer functions: a) Linear threshold function b) Piecewise linear function c) Sigmoid function d) Gaussian function.

$$g(y) = \begin{cases} 0 & \text{if } y < t \\ 1 & \text{if } y \geq t \end{cases} \tag{1.2}$$

and f is called a *weighted linear threshold function.* The second is the *continuous model* with $S = [0, 1]$. In this case, g is typically a monotone increasing function such as the *sigmoid function* (see Figure 1.2c) given by

$$g(y) = \frac{1}{1 + a^{-(by)}} \tag{1.3}$$

for $a, b \in R^+$. The continuous model is popular because it is easier to construct. The discrete model is popular because its behavior is easier to analyze (however it uses more hardware). A neural network is characterized by the network topology, the connection strength (i.e. weights) between pairs of neurons, the neurons properties (i.e. transfer functions) and the learning algorithms.

Artificial neural networks can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neurons' outputs and neurons' inputs. Based on the interconnection pattern (architecture), artificial neural networks are grouped in two categories: *feed-forward networks* (in which graphs have no loops and cycles) and *feed-back* (or *recurrent*) *networks* (in which loops or cycles occur because of feed-back connections). Different network topologies yield different network behaviors and also require appropriate learning algorithms. Figure 1.3 illustrates the two types of network topologies.

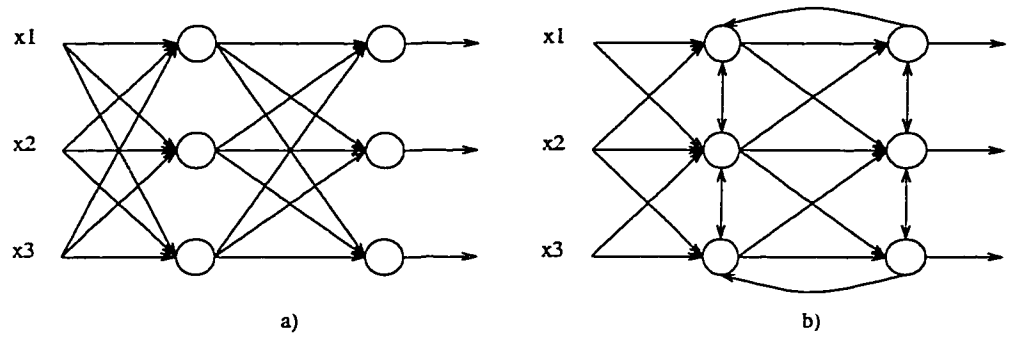


Figure 1.3: a) Examples of neural networks architectures: Feed-forward network b) Feed-back network.

A learning process in artificial neural networks is the problem of updating the connection weights so that the network can efficiently perform a specific task. Learning can also be viewed as the problem of minimizing an objective function which is the error between the network output and the desired output. Efficient learning algorithms for specific topologies have been proposed in literature.

1.2 Multiple-valued logic

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 $P_k = \bigcup_{n \geq 1} P_k^n$ is the set of all k-valued logic functions.

Multiple-valued logic has been the subject of much research over many years [Epstein 74, Hurst 84, Smith 81]. Besides reduction in chip area and interconnections, multiple-valued logic offers other benefits such as potential for image processing [Kameyama 87], and speech recognition [Smith 88]. The development of multiple-valued logic algebras has also proceeded since the pioneering works of Lukasiewicz [Lukasiewicz 20] and Post [Post 21]. [Post 21] presented the first functionally complete set as basic operations in multiple-valued logic. Since then there have been many multiple-valued logic algebras developed [Lu 85, Tang 89, Vranesic 70, Ngom 95a, Ngom 97a, Ngom 97b, Ngom 95b] and, applications for image processing network [Kameyama 87] and knowledge information processing systems [Hanyu 89] presented.

1.3 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. 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 $f : S^n \subseteq R^n \mapsto K$. 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 [Chan 88] and since then various other models have been described [Obradović 96, Tang 95, Watanabe 90].

1.4 Research statement

The problem we address in this thesis is that of implementing multiple-valued logic functions by neural networks. In particular we discuss interest-

ing models of multiple-valued logic neurons (and neural networks) and how they might be used to learn and compute multiple-valued logic functions. In most models, we will be interested in optimizing some identified resource of space or time such as number of thresholds, number of neurons or size of strip. We will also be interested to study the computational and learning abilities of our main model of neuron: the *multiple-valued multiple-threshold perceptron.*

1.5 Existing methods

[Chan 88] first developed a model of multiple-valued logic neural network (see section 2.1). In that model, neurons simulate *Kleenean functions* —a special class of three-valued logic functions— using the well known back-propagation learning algorithm. The model can be extended to make probabilistic predictions. It can also be used to make decisions with fuzzy input.

A design of multiple-valued logic adaptive linear neuron —adaline— is described in [Watanabe 90] (see section 2.2). Such adaline contains multiple-valued logic operations in place of the usual linear combiner and thus has a faster operation speed and a more robust separation ability. The model uses a modified adaline learning algorithm to learn two-valued logic functions with noisy inputs.

[Cao 93, Tang 93] defined a continuous multiple-valued neural network containing piecewise linear units as processing elements (see section 2.3). A multiple-valued logic algebra with only three basic arithmetic operations is introduced and is proven to be functionally complete. Several logic functions can thus be synthesized using the algebraic system. The authors also described a learning technique, which is largely borrowed from back-propagation training algorithm, to learn any multiple-valued logic function.

[Tang 95] described a learning multiple-valued logic network based on back-propagation (see section 2.4). The learning network is derived directly from a canonical realization of multiple-valued logic functions and therefore its functional completeness is guaranteed.

It has been shown in [Obradović 92] that continuous neurons with limited precision are essentially equivalent to discrete multiple-valued neurons (see section 2.5). Such model of neurons uses a class of multiple-valued multiple-threshold functions as transfer functions and hence generalizes the discrete binary perceptron model which uses the linear threshold function. The perceptrons studied in this thesis belong to the more general class of multiple-valued multiple-threshold perceptrons and in [Obradović 90a] it is

given the first instance of learning algorithm for multiple-valued multiple-threshold perceptrons.

Some results and methods are known in literature concerning the synthesis of two-valued logic functions by feed-forward multilayer neural networks composed of discrete binary perceptrons [Duda 73, Littlestone 89, Littlestone 87, Minsky 69, Parberry 88, Siu 95]. [Minsky 69] is the first who gave necessary and sufficient conditions for two-valued logic functions to be linearly separable, that is, to be learned and hence simulated by discrete two-valued perceptrons.

Two-valued logic perceptrons are very closely related to *threshold logic elements* which have long been investigated by many authors [Hampel 71, Muroga 71]. Perceptrons can be thought of as threshold logic elements with learning capability. A threshold logic element forms a functionally complete set by itself and thus is used as building block to construct complexe logic circuits implementing logic functions. Therefore comes the important problem of minimizing the size or depth of such threshold circuits.

Extensions of the two-valued threshold logic elements to the *two-valued multiple-threshold logic elements* were also introduced and analyzed in literature [Haring 65, Krueger 86, Olafsson 88, Takiyama 85]. It is known that any two-valued logic function can be implemented by a single such element. More general circuit elements called *multiple-valued multiple-threshold elements* have been studied [Abd-El-Barr 86b, Aibara 72, Ishizuka 77, Sasao 89].

Interesting new methods for synthesis of multiple-valued logic functions have been described in literature. These new approaches to synthesis, based on natural or physical laws, were introduced mostly to search for a minimal circuit representation (or equivalently, a minimal logic expression) of a given multiple-valued logic function. The only known algorithm for finding minimal multiple-valued logic expressions is exhaustive search. The excessive computation time makes this approach impractical. Especially, multiple-valued sum-of-products expressions are interesting because of the ease with which they can be implemented by programmable logic arrays [Bender 85, Sasao 89]. Because of the computational complexity associated with minimal sum-of-products solutions, there is considerable interest in heuristics. Typical heuristics is that, first a minterm is selected and then an implicant is chosen that covers the minterm [Besslich 86, Dueck 92a, Fei 93, Miller 94b, Muroga 79, Muzio 86, Sasao 89, Tirumalai 91, Yang 90]. This process is repeated until the given expression is covered. [Dueck 92b, Yildirim 93, Yurchak 90] proposed multiple-valued logic design methods which employ *simulated annealing* [Kirkpatrick 83]. [Hata 94, Kaczmarek 95] proposed neural network techniques. [Hata 97, Miller 94a, Wang 96, Zaitseva 96] pro-

posed solutions using *genetic algorithms*. [Kabakçioglu 90, Lloris-Ruiz 93, Roman-Roldan 92] used *information theoretic* (entropy) approaches to minimization of logic expressions.

1.6 Motivations

[Cao 93, Chan 88, Tang 93, Tang 95, Watanabe 90] do not discuss how to construct minimal neural networks from their respective models. In all of these papers, the size and depth of the networks is fixed in advance. This is a major drawback since many logic functions can be synthesized with minimal size or minimal depth networks.

The homogeneous multiple-valued perceptron learning algorithm presented in [Obradović 90a] has the weakness of being able to learn only a very tiny portion of the set of separable functions, so it has a low capacity. Also, unlike the binary perceptron learning algorithm, it is not known how many logic functions can be learned by the homogeneous learning algorithm. The number of logic functions that can be learned by a single neuron is called the *capacity* of the neuron (or, of the learning algorithm). There are no known results on the capacity of homogeneous perceptrons and also of the general class of multiple-valued multiple-threshold perceptrons. Also, no learning algorithm is known in general for the multiple-valued multiple-threshold perceptrons.

Multiple-valued multiple-threshold perceptrons are multiple-valued multiple-threshold logic elements with learning abilities. A minimal multiple-valued logic perceptron which computes a given logic function is a perceptron containing the least number of thresholds. The problem of finding such minimal perceptron for a function is difficult and is still left open.

Finding a minimal logic expression for a given function is very difficult problem. The simulated annealing [Dueck 92b, Yildirim 93, Yurchak 90], genetic algorithms [Miller 94a, Wang 96, Zaitseva 96] and neural networks [Hata 94, Kaczmarek 95] based approaches to expression minimization introduced in literature have the tendency to produce *local optimum* solutions. These are still very good solutions, however it seems to us that better solutions can be obtained by improving there methods.

1.7 Contributions

Our contributions are sumerized as follows.

- We obtain bounds on the capacity of multiple-valued multiple-threshold perceptrons (in [Ngom 99d, Ngom 99e]).
- We describe learning algorithms for permutably homogeneous multiple-valued multiple-threshold perceptrons (in [Ngom 98c]).
- We introduce two techniques of constructing multiple-valued logic neural networks given any function (in [Ngom 99b]).
- We propose a genetic algorithm method to find minimal multiple-valued multiple-threshold perceptrons (in [Ngom 98b]).
- We provide a *neuro-genetic* minimization approach which searches for minimal multiple-valued logic neural network architectures (in [Ngom 99a]).

1.8 Organization of the document

In chapter 2, we make a survey on currently known models of multiple-valued neural networks. In chapter 3, we introduce our model of multiple-valued neuron, the *multiple-valued multiple-threshold perceptron*, and define many (new) concepts that are used throughout the thesis. Our main research is based on this model. In chapter 4, based on well known relationships between linear partitions and minimal pairs, we derive formulae for the number of multilinear partitions of a point set in general position and of the set K^2. We obtain results on the capacity of a single (n, k, s)-perceptron, respectively for $V \subset R^n$ in general position and for $V = K^2$. Finally, we describe a fast polynomial-time algorithm for counting the multilinear partitions of K^2. In chapter 5, the learning abilities of single (n, k, s)-perceptrons are examined. 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 homogeneous k-valued logic function given as input. In chapter 6, we consider the problem of synthesizing multiple-valued logic functions by neural networks. A genetic algorithm which finds either the longest strip or the maximum separable subset for a subset in K^n, by using different fitness functions, is described. A strip contains points located between two parallel hyperplanes. Repeated application of the genetic algorithm partitions the space K^n 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 logic function. Preliminary experimental results are presented and compared with other methods such as *tabu search* [Glover 93]. In chapter 7, we address the problem of minimizing multiple-valued perceptron. Every n-input k-valued logic function can be implemented using a (n, k, s)-perceptron, for some number of thresholds s. We propose a genetic algorithm to search for an optimal (n, k, s)-perceptron that efficiently realizes a given multiple-valued logic function, that is to minimize the number of thresholds. In chapter 8, we propose a neuro-genetic approach to the minimization of multiple-valued logic sum-of-product expressions.

Chapter 2

Literature Review

2.1 Neural logic networks

Artificial neural networks are oftens criticized for lacking the symbolic reasoning and semantic representation found in rule-based expert systems. Instead of using numerical values, the rule-based approach relies on a symbol system to represent the human problem-solving process in the form of procedural or heuristic rules. The chaining of rules under a set of input conditions solves problems. While rich in knowledge representation and reasoning, expert systems are unable to learn and adapt to new users' needs. And upon receiving unexpected input, expert systems, unlike neural networks, cannot respond and may fail abruptly.

National University of Singapore researchers [Chan 88] have proposed a neural network known as the *neural logic network* or simply *neulonet* (NLN), which combines the strengths of neural networks and expert systems. NLN differs from other neural networks models by having an ordered pair of numbers associated with each node and connection, as shown in Figure 2.1. Let Q be the output node and $P_1, \ldots, P_n$ be input nodes. Also, let value associated with node P_i be denoted by $x_i = (a_i, b_i) \in \{0,1\}^2$, and the weight for the connection from P_i to Q be $w_i = (\alpha_i, \beta_i) \in R^2$. Each node's ordered pair takes one of three values – (0, 0) for *unknown*, (0, 1) for *false*, or (1, 0) for *true*; (1, 1) is undefined. Let $K = \{(0,0),(0,1),(1,0)\}$, $\vec{x} = (x_1, \ldots, x_n) \in K^n$ and $\vec{w} = (w_1, \ldots, w_n) \in R^{2n}$. Neuron Q computes a simple function of the form $f_{\vec{w}} : K^n \mapsto K$ and

$$f_{\vec{w}}(\vec{x}) = g(\sum_{i=1}^{n}(\alpha_i a_i - \beta_i b_i)) \tag{2.1}$$

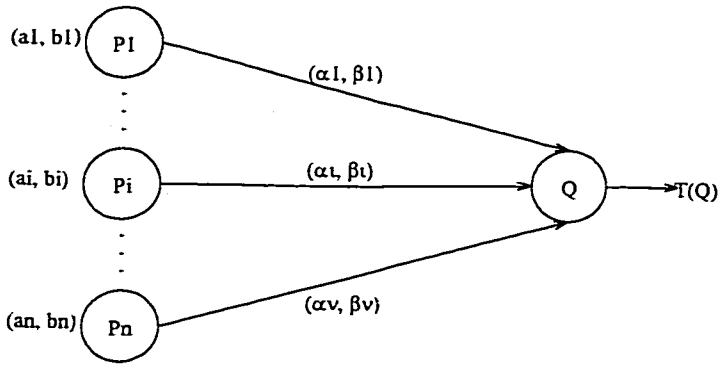


Figure 2.1: Basic neural logic network.

for some transfer function $g : R \mapsto K$ given by

$$g(y) = \begin{cases} (0,1) & \text{if } y \leq -t \\ (0,0) & \text{if } -t < y < t \\ (1,0) & \text{if } t \leq y \end{cases} \tag{2.2}$$

where the threshold value t is usually set to 1.

NLNs are three-valued logic neural networks used to represent a set of non-recursive propositional rules and to infer the truth values of the unknown propositions. A proposition Q is represented as a NLN output neuron (labelled 'Q' within a circle) and the truth value of Q, denoted by $T(Q)$, is given by the neuron's output. The connection weight from a neuron denoting proposition P_i to a neuron denoting proposition Q can be viewed as the strengths of support (α_i) and opposition (β_i) respectively of proposition P_i for proposition Q.

Applying (2.2), it is easy to verify that the truth values given in Table 2.1 are reproduced by the NLNs given in Figures 2.2. It should be noted that the underlying algebra is not Boolean, instead it is either a *Kleene*'s or *Bochvar*'s algebra, depending on the interpretation of value (0, 0) — *unknown* in Kleene's logic or *meaningless* in Bochvar's logic.

Training is an important feature of all neural networks and NLN uses the common *back-propagation* training method. NLNs are also extended into probabilistic NLNs [Teh 89] and fuzzy NLNs [Teh 90c]. In either way, a neuron's value is generalized to pair $(a, b) \in [0, 1]^2$ (instead of $(a, b) \in \{0, 1\}^2$) with $0 \leq a + b \leq 1$. In a probabilistic NLN a and b represent respectively the probability of a proposition to be true and false. Thus the probability of unknown will be $1 - a - b$. In a fuzzy NLN, a denotes the amount of

A	B	A AND B	A OR B	A XOR B	NOT A
(1, 0)	(1, 0)	(1, 0)	(1, 0)	(0, 1)	(0, 1)
(1, 0)	(0, 1)	(0, 1)	(1, 0)	(1, 0)	
(1, 0)	(0, 0)	(0, 0)	(1, 0)	(1, 0)	
(0, 1)	(1, 0)	(0, 1)	(1, 0)	(1, 0)	(1, 0)
(0, 1)	(0, 1)	(0, 1)	(0, 1)	(0, 1)	
(0, 1)	(0, 0)	(0, 1)	(0, 0)	(0, 0)	
(0, 0)	(1, 0)	(0, 0)	(1, 0)	(1, 0)	(0, 0)
(0, 0)	(0, 1)	(0, 1)	(0, 0)	(0, 0)	
(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	

Table 2.1: Some basic operators in Kleene's logic.

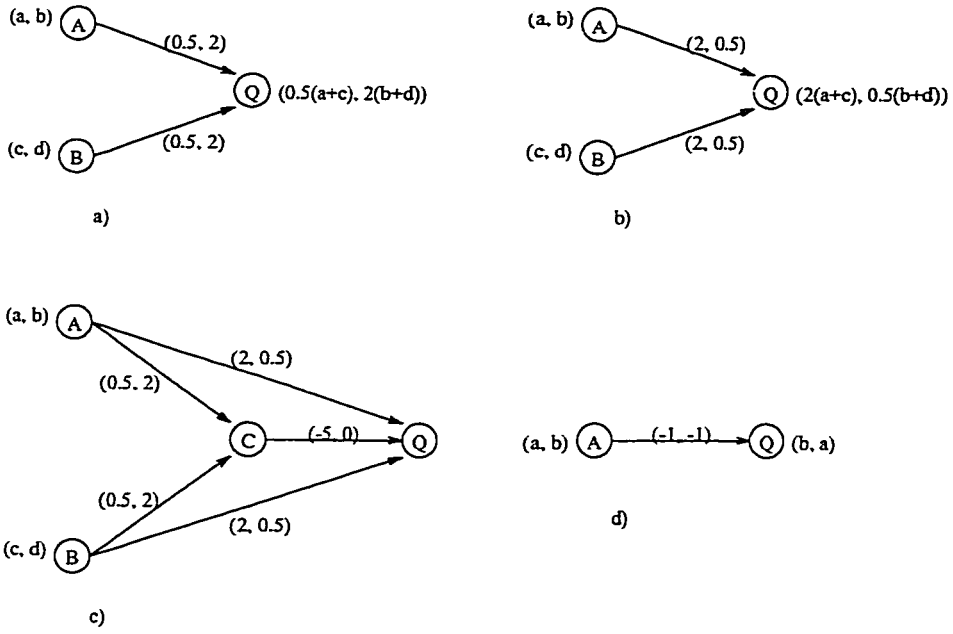


Figure 2.2: Computations of basic Kleenean functions: a) AND b) OR c) XOR d) and NOT functions.

evidence that is for the condition represented by the node while b denotes the amount of evidence against the condition. Thus the quantity $1 - a - b$ will express the lack of evidence regarding the condition.

2.2 Adaptive linear neurons with multiple-valued logic operations

[Watanabe 90] presents a design of a neuron model they called *multiple-valued logic neuron*, in which multiple-valued logic operations are used to produce analog responses to be fed to a quantizer. The proposed multiple-valued logic neuron model is trainable to classify the input pattern vectors according to the special inputs called *desired responses*. The advantages of the multiple-valued logic neuron model over the ordinary linear neuron model are a robust separation ability and a very fast operation speed in pattern recognition due to the employment of multiple-valued logic operations. The complexity in the training algorithm is similar to that of the ordinary adaptive linear neuron model also called *adaline*.

For easy introduction of the multiple-valued logic neuron, the features of the ordinary Adaline model is briefly introduced first, according to [Widrow 88]. Given a weight vector $\vec{w} \in R^n$ and an input vector $\vec{x} \in \{-1, 0, 1\}^n$ an adaline computes a simple function of the form $f_{\vec{w}} : \{-1, 0, 1\}^n \mapsto \{-1, 1\}$ and

$$f_{\vec{w}}(\vec{x}) = g(\vec{w}\vec{x}) \tag{2.3}$$

where the transfer function $g : R \mapsto \{-1, 1\}$ is a linear threshold funtion. However, unlike the perceptron, the adaline has an adaptive threshold level t which is given as follows

$$t = \frac{t_1 + t_2}{2} \tag{2.4}$$

where t_1 is the latest weighted sum adjusted for the non-desired input patterns responded incorrectly, and, t_2 is the latest weighted sum adjusted for the desired input patterns.

Now we give the definition of the multiple-valued adaline. Let $I = [0, r]$, with $r > 0$. Given a weight vector $\vec{w} = (w_1, \ldots, w_n) \in I^{2^n}$ (with $w_i \in I^2$) and an input vector $\vec{x} \in \{-1, 0, 1\}^n$, a multiple-valued logic adaline computes a simple function of the form $f_{\vec{w}} : \{-1, 0, 1\}^n \mapsto \{-1, 1\}$ and

$$f_{\vec{w}}(\vec{x}) = g(\bigwedge_{i=1}^{n} (\alpha(w_i)x_i)) \tag{2.5}$$

whose transfer function $g : [-r, r] \mapsto \{-1, 1\}$ is a linear threshold function. The operation $\wedge$ is the well-known multiple-valued logic *minimum* operation which returns the minimum of its arguments. Unlike the ordinary

adaline, the weights here are pairs of positive numbers $w_i = (w_{N_i}, w_{P_i}) \in I^2$ for $1 \leq i \leq n$, and where $\alpha : I^2 \mapsto [-r, r]$ is defined by

$$\alpha(w_i) = \begin{cases} -w_{N_i} & \text{if } x_i < 0 \\ w_{P_i} & \text{if } x_i > 0 \end{cases} \tag{2.6}$$

and $\alpha(w_i)x_i = w_{N_i} \wedge w_{P_i}$ for $x_i = 0$.

The multiple-valued logic adaline has also an adaptive threshold level t which is calculated as follows

$$\begin{array}{rcl} t_1 & = & \bigvee_{i=1}^{n} w_{N_i} \vee \bigvee_{i=1}^{n} w_{P_i} \\ t_2 & = & \bigwedge_{i=1}^{n} w_{N_i} \wedge \bigwedge_{i=1}^{n} w_{P_i} \\ t & = & \frac{t_1 + t_2}{2} \end{array} \tag{2.7}$$

where $\vee$ is the multiple-valued logic *maximum* operation which returns the maximum of its arguments.

The main difference between the ordinary adaline and the multiple-valued logic adaline is that the multiple-valued logic adaline uses $\vee$ and $\wedge$ operations to produce an analog response and a binary output. Equations (2.3), (2.4), (2.5) and (2.7) illustrate such difference. There are some comments we want to raise about such multiple-valued logic neuron model as described in [Watanabe 90]. First, such model does not match our definition of multiple-valued logic neuron as stated in chapter 1.3. Their multiple-valued logic adaline is a binary neuron like the ordinary adaline. Even though the input values are taken from the set $\{-1, 0, 1\}$, the multiple-valued logic adaline can only simulate a binary Boolean logic function whose input vector may contain some *noise* (given by value 0). Also, it does not compute three-valued logic functions, such as Kleenean logic functions for instance (see section 2.1), since a three-valued logic neuron would separate the input vectors into three classes whereas the multiple-valued logic adaline always partitions the input vectors into two classes. Second, it is not truly clear in [Watanabe 90] whether r is a real or an integer number. If r is a real, then the operations $\vee$ and $\wedge$, they labelled *multiple-valued*, are respectively *maximum* and *minimum* operations over the interval $[-r, r] \subset R$. And thus it does not make their adaline (with $\vee$ and $\wedge$ operations) multiple-valued.

2.3 Synthesis of multiple-valued logic functions using piecewise linear units networks

[Cao 93, Tang 93] describes a model of analog neural network used to simulate multiple-valued logic functions. Let $K = \{0, \ldots, k-1\}$. Let $F \subseteq P_k$,

then F is *functionally complete* (or *complete*, for short) in P_k if and only if every function of P_k can be constructed by compositions of functions from F only. For instance, the sets {AND, OR, NOT} and {NOR} constitute two functionally complete sets in P_2 since any function in P_2 can be realized by operators from either one of these two sets. A most practically useful and well known complete set in P_k is the set $S = \{\vee, \wedge, {}^a x^b\}$, where the unary operator ${}^a x^b$ is the multiple-valued logic *window literal* function defined as follows

$$ {}^a x^b = \begin{cases} k-1 & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \tag{2.8} $$

and $a, x, b \in K$.

For example, every multiple-valued logic function $f \in P_k^n$ can be expressed using the following canonical representation form

$$ f(\vec{x}) = \bigvee_{i=1}^{m} (c_i \wedge {}^{a_{i1}} x_1^{b_{i1}} \wedge \cdots \wedge {}^{a_{in}} x_n^{b_{in}}) \tag{2.9} $$

where c_i, a_{ij}, b_{ij} are constant over K, $1 \leq i \leq m \leq k^n$, $1 \leq j \leq n$, and $\vec{x} \in K^n$.

The main idea in [Tang 93] is to define a model of neuron to be used to realize each operator in S, and then for any given multiple-valued logic function, construct a multiple-valued neural network derived directly from the canonical representation given in (2.9). In such multiple-valued logic neural network, each neuron computes a simple function of the form $f_{\vec{w}} : R^n \mapsto R$ and

$$ f_{\vec{w}}(\vec{x}) = g(\vec{w}\vec{x}) \tag{2.10} $$

whose transfer function, $g : R \mapsto R$, is a *piecewise linear function* defined as

$$ g(y) = \begin{cases} 0 & \text{if } y < t \\ y & \text{if } y \geq t \end{cases} \tag{2.11} $$

where the threshold $t \in R$ and $\vec{w}, \vec{x} \in R^n$.

As it is shown in Figures 2.3-2.4, basic multiple-valued logic neural networks can be designed to simulate $\vee$, $\wedge$ and ${}^a x^b$ operators (the numbers inside the circles are the threshold levels). Therefore, by direct synthesis (i.e. canonical representation in formula (2.9)), one can simulate every

multiple-valued logic function using these basic networks. For instance, Figure 2.5 shows the directly synthesized network realization of the four-valued function defined in Table 2.2.

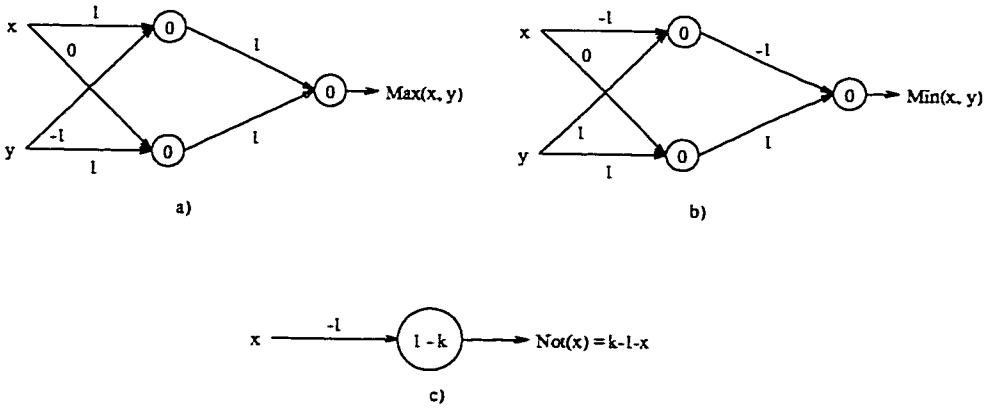


Figure 2.3: Simulations of multiple-valued logic a) maximum b) minimum and c) negation functions.

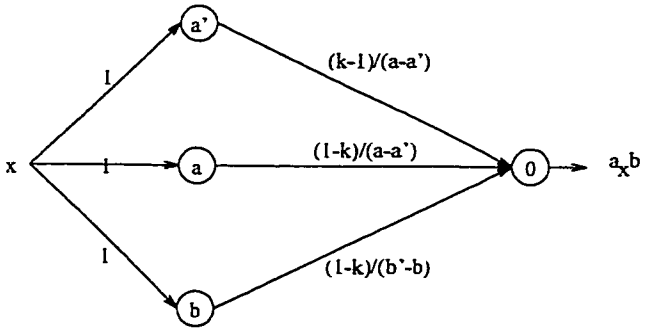


Figure 2.4: Simulation of multiple-valued logic window literal function ($a' < a < b < b'$).

Network construction with direct synthesis is clearly unrealistic for most functions. Although some techniques may produce a minimized realization, no method short of an exhaustive search is available for guaranteeing a minimum representation. Furthermore, S may not be an optimum complete set itself for many logic functions.

However, a piecewise linear units network with learning capability may provide a powerful approach to multiple-valued logic functions synthesis.

$x_2 \setminus x_1$	0	1	2	3
0	0	1	2	3
1	1	2	3	2
2	2	3	2	1
3	3	2	1	0

Table 2.2: Truth table of some two-place four-valued logic function.

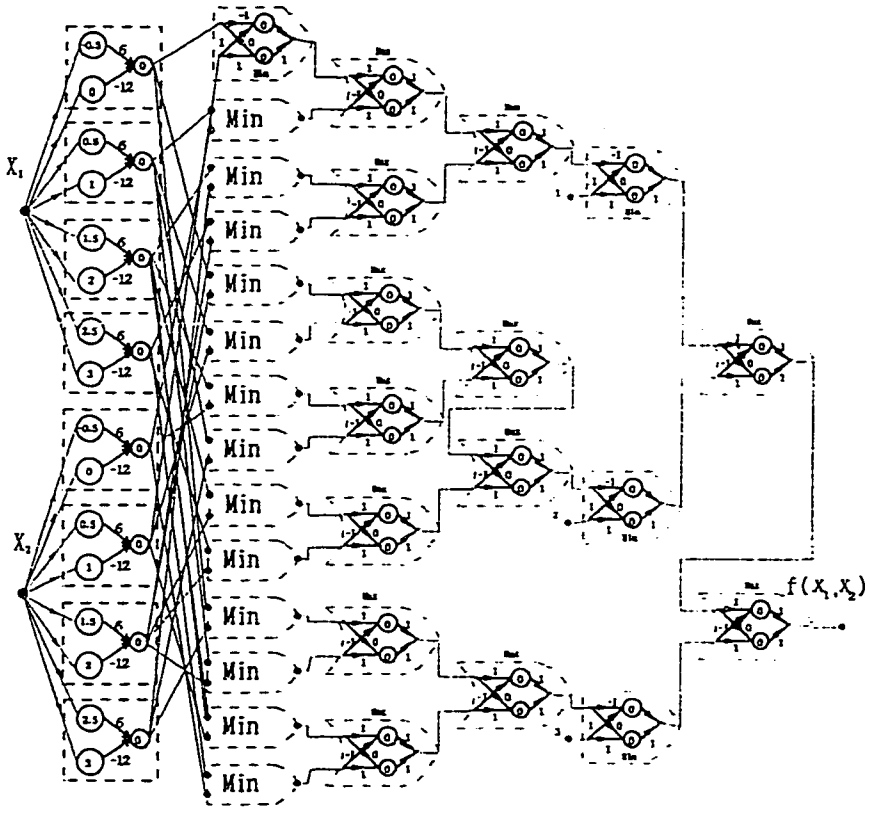


Figure 2.5: Direct synthesis network for Table 2.2.

For example, the function given in Table 2.2 can be realized with a two-layer network as shown in Figure 2.6. The function is implemented with appropriate selection of weights and thresholds.

[Cao 93] developed a gradient-based least-square error learning algo-

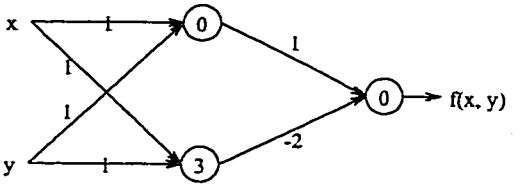


Figure 2.6: Piecewise linear units network for Table 2.2.

rithm to minimize the error between actual and desired outputs. Their algorithm combines back-propagation learning with other features of multiple-valued logic networks, including the piecewise linear node function and the multiple-valued representation. The authors applied the algorithm to learn $\vee$, $\wedge$, ${}^{a}x^{b}$ and some two-place four-valued logic functions. Even though convergence did occur for these functions, nowhere in [Cao 93] did the authors mention how many layers and piecewise linear nodes per layer were required for each logic function in order to ensure good convergence. This make us wonder how the network depth and size vary with k and n together. A good interesting problem here would be to design a technique to construct a depth-optimal, or size-optimal or depth&size-optimal network for any given multiple-valued logic function to be learned.

2.4 Multiple-valued logic networks with learning capability

[Tang 95] describes a multiple-valued logic network, with learning capability, based on back-propagation. The learning multiple-valued logic network is derived directly from a canonical realization of multiple-valued logic functions and therefore its functional completeness is guaranteed. The authors extend the traditional back-propagation to include the prior human knowledge on multiple-valued logic networks, for example, the architecture and the number of nodes and layers. The prior knowledge from the multiple-valued logic canonical form can be used as initial parameters of the multiple-valued logic network in its learning process. As a result, the prior knowledge can guide back-propagation learning process to get started from a point in the parameter space that is not far from a global optimum.

Multiple-valued logic networks are used in many application such as image processing [Kameyama 87] and speech recognition [Smith 88]. However, these networks cannot adapt to change in their environments. By adding learning ability in these networks, one can design systems that can adjust to

environment changes and get their power from their ability to learn, and, on the other hand, from their capacity to be modified and extended. Designing a multiple-valued logic network can be based on learning a relationship that transforms inputs to outputs given a set of examples and forcing the network to yield a particular response to a specific input. Learning is also necessary when some information about inputs/outputs is unknown or incomplete a priori, so that no design of a network can be perform in advance.

The key idea in [Tang 95] is to construct a feed-forward multiple-valued logic network based on a canonical realization of multiple-valued logic functions, and to conduct the learning in manner analogous to back-propagation. It is well known that any multiple-valued logic function can be represented in a canonical $\vee$-of-$\wedge$ form

$$f(\vec{x}) = \bigvee_{i=1}^{m} (f(e_{i1}, \ldots, e_{in}) \wedge {}^{e_{i1}}x_1^{e_{i1}} \wedge \cdots \wedge {}^{e_{in}}x_n^{e_{in}}) \tag{2.12}$$

where the e_{ij}'s are constant over K, $1 \leq i \leq m \leq k^n$, $1 \leq j \leq n$, and $\vec{x} \in K^n$. The corresponding architecture is shown in Figure 2.7, where the node functions in the same layer are of the same type.

Consider a simple example of a two-variable four-valued logic function shown in Table 2.3. A canonical realization of the function can be the $\vee$ of three $\wedge$-terms

$y \setminus x$	0	1	2	3
0	1	0	0	0
1	0	0	0	0
2	0	0	3	0
3	2	0	0	0

Table 2.3: Truth table of some two-place four-valued logic function.

$$f(x,y) = (1 \wedge {}^{0}x^{0} \wedge {}^{0}y^{0}) \vee (2 \wedge {}^{0}x^{0} \wedge {}^{3}y^{3}) \vee (3 \wedge {}^{2}x^{2} \wedge {}^{2}y^{2}) \tag{2.13}$$

whose corresponding multiple-valued logic network is shown in Figure 2.8.

Each node l in layer 1 is a window literal unit and its node function is given by ${}^{a_{ij}}x_{j_l}{}^{b_{ij}}$, where $a_{ij} \leq b_{ij}$, $1 \leq l \leq nm$, $1 \leq i \leq n$ and $1 \leq j \leq k$. The window literal function for the node is shown in Figure 2.9. As the values of a_{ij} and b_{ij} change, the literal function varies accordingly, thus exhibiting

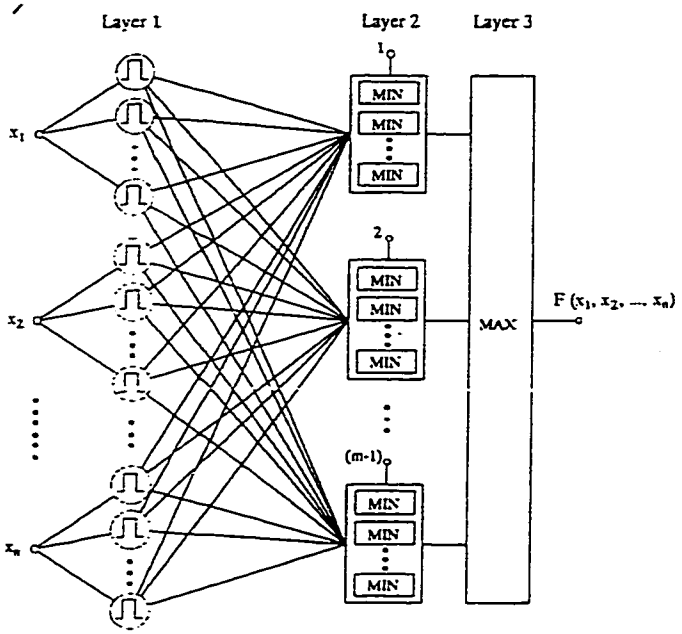


Figure 2.7: Learning multiple-valued logic network architecture based on a canonical realization of multiple-valued logic functions.

various form of literal functions. In n-variable k-valued case, at most nk^n literal nodes are required in order that each input vector can be individually selected. In order to do learning with gradient descent, derivatives of the literal functions are needed. The derivatives of a literal function do not exist at $x = a_{ij}$ and $x = b_{ij}$. A literal function ${}^a x^b$ can be seen as a *double-threshold function* (with thresholds a and b) and thus can be replaced by the use of the following two sigmoid functions

$$ {}^a x^b = (k-1)[1 - sigmoid(a-x)] = (k-1)\frac{1}{1+e^{\lambda(a-x)}} \tag{2.14} $$

and

$$ {}^a x^b = (k-1)sigmoid(b-x) = (k-1)\frac{1}{1+e^{-\lambda(b-x)}} \tag{2.15} $$

where $\lambda \to +\infty$. These functions are differentiable and saturate to 0 or $k-1$ at both extreme.

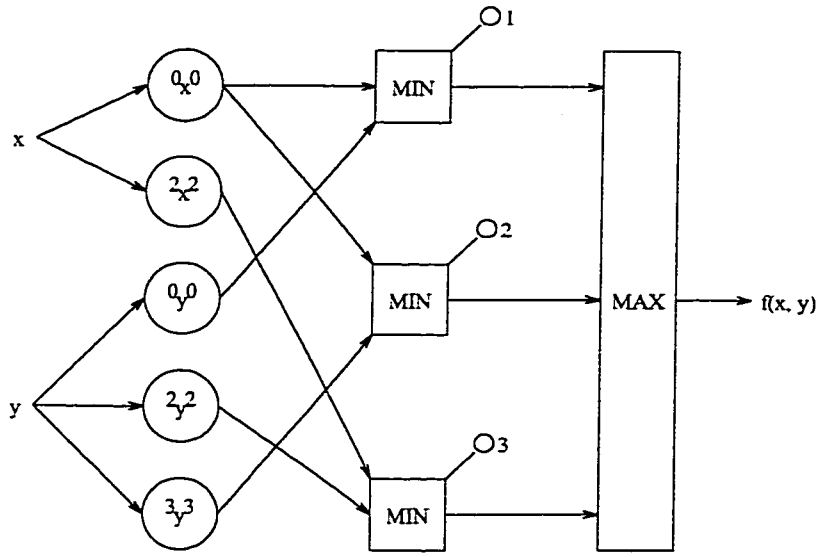


Figure 2.8: Multiple-valued logic network for the function of Table 2.3.

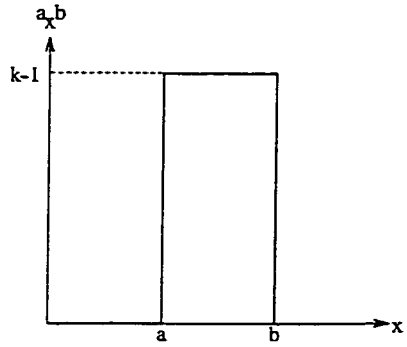


Figure 2.9: Multiple-valued logic window literal function.

Each node i in layer 2 corresponds to a minimum function $\wedge$. Each node selects a particular area of the function and defines its function value by means of a constant c_i included within the $\wedge$-term. There are at most k^n *MIN* nodes and the function is given by

$$MIN_i = c_i \wedge \bigwedge_{j=1}^{n} a_{ij} x_j^{b_{ij}} \tag{2.16}$$

where $1 \leq i \leq m \leq k^n$. Derivatives of the *MIN* functions are also needed

in order to do learning with gradient descent. Since these functions are not differentiable, some heuristic approximation of the derivatives are developed in [Tang 95].

The unique node in layer 3 (i.e. the output node) is a maximum operator $\vee$ between the $\wedge$-terms

$$O = \bigvee_{i=1}^{m} MIN_i \tag{2.17}$$

where $1 \leq m \leq k^n$. Approximate derivatives are also defined for this function.

Unlike most conventional neural networks, this network has no weights. Furthermore, modifiable parameters are present only on layer 1. The method used to train this network is the back-propagation algorithm in which the parameters a_{ij} and b_{ij} of the literal nodes are modified so as to minimize a cost function E. Since back-propagation employs gradient descent thus a_{ij} and b_{ij} are updated in accordance with the rules:

$$\Delta a_{ij} = -\eta \frac{\partial E}{\partial a_{ij}} \text{ and } \Delta b_{ij} = -\eta \frac{\partial E}{\partial b_{ij}} \tag{2.18}$$

where $0 < \eta \leq 1$ is the learning rate. One important quality of the model is that the network training is very fast since it involves only the first layer (the constants c_i's in layer 2 are fixed and the output layer has no parameters). However a problem here is that, for a given multiple-valued logic function, a (near) minimal network must first be realized for it and then apply back-propagation algorithm on the network to find the window literal parameters. Finding minimal canonical representation for a given function is very difficult in general. A representation has at most k^n $\wedge$-terms containing each n literal operators. There exists no efficient minimization algorithm that does better than exhaustive search. One way to overcome this problem is to let the network construct itself. That is, starting from a (near optimal) canonical network, some literal (or *MIN*) nodes or some connections between layers can be removed from the initial network during the learning process. The removal mechanism can be done using some other optimization method such as *evolutionary algorithms*. The parameters c_i's of layer 2 are not necessarely fixed and thus can be learned either by the back-propagation algorithm or by the evolutionay algorithm. This way we let the network determine its optimal size. As far as we know, this idea is original and we give a more detailed description of our method in section 2.4.

2.5 Analog neural networks of limited precision

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 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.

[Obradović 92] studied analog (i.e. continuous) neurons of limited precision and it has been shown that they are essentially discrete multiple-valued logic neurons. For practical purposes, the restriction of limited precision on continuous neurons appears to be reasonable assumption (provided that the precision is not too small). For instance, let $g : R \mapsto [0,1]$ and $K = \{0,\ldots,k-1\}$. It is reasonable to suppose that a limited precision device which approximates a real-valued function g actually computes some function with range the k rational values $R_k = \{\frac{i}{k-1} | i \in K, 0 \leq i < k\}$. This is sufficient for all practical purposes provided k is large enough. Since R_k is isomorphic to K, one can formally define the limited precision variant of g to be the function $g_k : R \mapsto K$ defined by

$$g_k(y) = round(g(y)(k-1)) \tag{2.19}$$

where $round : R \mapsto N$ is the natural rounding function defined by $round(y) = n$ if and only if $n - 0.5 \leq y < n + 0.5$. It was shown in [Obradović 92] that if a neuron transfer function is monotone increasing, then an analog neuron of limited precision can be represented as a *discrete homogeneous k-valued neuron.* More formally, given an input vector $\vec{x} \in K^n$, a weight vector $\vec{w} \in R^n$ and a threshold vector $\vec{t} \in R^{k-1}$, a homogeneous neuron computes a *weighted multilinear threshold function* of the form $f_{\vec{w}} : K^n \mapsto K$ and

$$f_{\vec{w}}(\vec{x}) = g_{\vec{t}}(\vec{w}\vec{x}) \tag{2.20}$$

for some transfer function $g_{\vec{t}} : R \mapsto K$, a *multilinear threshold function*, defined as

$$g_{\vec{t}}(y) = \begin{cases} 0 & \text{if } y < t_1 \\ i & \text{if } t_i \leq y < t_{i+1} \text{ for } 1 \leq i \leq k-2 \\ k-1 & \text{if } t_{k-1} \leq y \end{cases} \tag{2.21}$$

where the t_i's are monotone increasing. The function $g_{\vec{t}}$ is called a *k-valued threshold function* and $f_{\vec{w}}$ is a *k-valued weighted threshold function.* Clearly, by analyzing homogeneous k-valued neurons, one is actually reasoning about certain aspects of behavior of monotone increasing continuous neurons of limited precision to $\log_2 k$ bits. For example, Figure 2.10 shows the monotone transfer function (in this case, a sigmoid) of a analog neuron, and the corresponding five-valued threshold function obtained by limiting its precision to the nearest multiple of 0.25.

In the homogeneous k-valued neural network model each neuron can have its own threshold vector. Let $\vec{o} = (o_0, \ldots, o_{k-1}) \in K^k$, then $\vec{o}$ is *monotone* if and only if $o_0 \leq o_1 \leq \cdots \leq o_{k-1}$ or $o_0 \geq o_1 \geq \cdots \geq o_{k-1}$. Repetitions of values in $\vec{o}$ can occur, for instance, $\vec{o} = (0,1,1,3,4,4)$, $\vec{o} = (8,8,8,8,6,5,4,3,3,0,0,0)$ and $\vec{o} = (2,2,2,2)$ are example of monotone vectors. [Obradović 96] considers an extension to a *heterogeneous k-valued neural network* model where, in addition, each neuron can have its own set of k monotone nondecreasing output values. That is, given a monotone output vector $\vec{o} \in K^k$, a heterogeneous k-valued neuron computes a function $f_{\vec{w}} : K^n \mapsto K$ where

$$f_{\vec{w}}(\vec{x}) = g^{\vec{o}}_{\vec{t}}(\vec{w}\vec{x}) \tag{2.22}$$

where each neuron's transfer function is a *generalized multilinear threshold function* $g^{\vec{o}}_{\vec{t}} : R \mapsto K$ defined by

$$g^{\vec{o}}_{\vec{t}}(y) = \begin{cases} o_0 & \text{if } y < t_1 \\ o_i & \text{if } t_i \leq y < t_{i+1} \text{ for } 1 \leq i \leq k-2 \\ o_{k-1} & \text{if } t_{k-1} \leq y \end{cases} \tag{2.23}$$

Although most of the neural network studies consider only monotone nondecreasing node transfer functions, the reasons for using them are not very well founded. In fact, it is known that some nonmonotone neural networks are quite promising both as computational and learning systems. For example, in the domain of control theory, good practical results are obtained using analog nonmonotone neurons model called *radial basis functions* [Moody 88]. In this model each neuron's transfer function is typically a *Gaussian function* (see Figure 1.2d). A powerful nonmonotone binary neural network model, called *alternating multilinear neural network* is studied

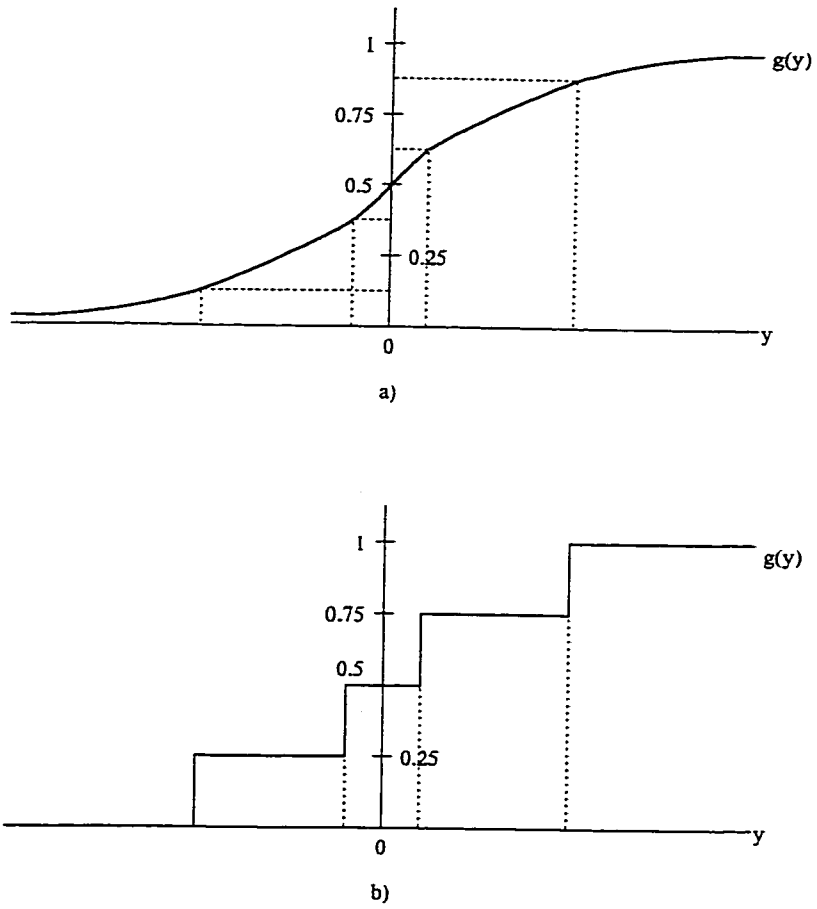


Figure 2.10: a) Sigmoid transfer function of a continuous neuron and b) the corresponding five-valued threshold function.

in [Olafsson 88, Takiyama 85]. In this model neurons compute *alternating weighted multilinear threshold functions* $f_{\vec{w}} : R^n \mapsto \{0,1\}$ where

$$f_{\vec{w}}(\vec{x}) = g_{\vec{t}}(\vec{w}\vec{x}) = \begin{cases} 0 & \text{if } t_{2i} \leq \vec{w}\vec{x} < t_{2i+1} \text{ for } 0 \leq i < \frac{s}{2} \\ 1 & \text{otherwise} \end{cases} \tag{2.24}$$

where $\vec{t} \in R^s$, $s \geq 2$. and the transfer function $g_{\vec{t}} : R \mapsto \{0,1\}$ is an *alternating threshold function.*

In [Obradović 96]. homogeneous and heteregeneous k-valued as well as alternating multilinear neural networks are extended to more general *non-*

monotone k-valued neural networks. Their neurons compute functions of the form of equations (2.22) and (2.23) in which the output vector $\vec{o}$ is not necessarily monotone. Nonmonotone k-valued neurons correspond to limited precision continuous neurons whose transfer function is not necessarily monotone.

Computational and learning abilities of all these neuron models are investigated in [Obradović 96, Obradović 94, Obradović 92] comparing them to previously studied monotone and nonmonotone binary models. Relationships between models are given, that is the resources of running time, size, depth and weight of some model needed to simulate another model. Resource trade-offs involving depth, size and weights among various models are also explored. The results of such studies indicate that significant savings are possible through selection of an appropriate nonmonotone multiple-valued model. Further research is needed to develop a general theory for identifying the optimal trade-off between complexity of neurons versus size and depth of a neural network designed using those neurons as its building blocks. Also, further research is needed to develop efficient learning methods for some identified k-valued neural networks (so far, learning algorithms exist only for the homogeneous k-valued neurons [Obradović 94]).

2.6 Genetic algorithms

In this section we briefly introduce the powerful optimization technique known as *genetic algorithms.* It will the main optimization tool used in all our optimization or search problems.

[Holland 75] 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 as shown in Figure 2.11 is commonly called the *Simple Genetic Algorithm* (or SGA, for short).

```
Procedure SGA;
    Set control parameters;
    Create a random initial population;
    Evaluate the initial population;
    Repeat
        Select individuals for next generation;
        Perform crossover and mutation;
        Evaluate the new population;
    Until Termination condition is reached;
```

Figure 2.11: Holland's simple genetic algorithm.

Using selection, crossover and mutation, the SGA creates the subsequent generation from the chromosomes of the current population. This generational cycle is repeated until a desired termination criterion is reached (for example, a predefined number of generations are processed or a convergence criterion is reached). Fundamental to the genetic algorithm are the control parameters, the fitness function, the genetic operators, the selection mechanism, and the mechanism to encode the solutions as chromosomes.

The encoding mechanism depends on the nature of the problem's variables. Choosing an appropriate problem representation is the first step in applying GA. If the problem involves searching a n-dimensional space then the representation of the problem is often solved by allocating a sufficient number of bits to each dimension to achieve the desired accuracy. In *pure* genetic algorithms, only binary chromosomes representations were allowed, but we can allow any problem representation that makes efficient computation possible.

The objective function, the function to be optimized, provides the mechanism for evaluating each chromosome. However, its range of value varies from problem to problem. To maintain uniformity over various problem domains, most GAs use the *fitness function* to normalize the objective function to a convenient range of 0 to 1. The normalized value of the objective function is the *fitness* of the chromosome, which the selection mechanism uses to evaluate the chromosomes of the population.

Selection models nature's *survival-of-the-fittest* mechanism. Fitter individuals in the current population survive while weaker ones perish. A fitter chromosome produces a higher number of offsprings than a weaker one, and thus has a higher chance of surviving in the subsequent generation. In the *fitness proportionate selection scheme* [Goldberg 89], an individual with fitness f_i is allocated an expected number $\frac{f_i}{\bar{f}}$ of offsprings, where $\bar{f}$ is the average fitness value of the population.

Crossover is the GA's crucial operation. Pairs of randomly selected chromosomes are subjected to crossover. In nature, the role of crossover is to juxtapose good genetic material called building blocks that already exist in two parents chromosomes to form a better fit child chromosomes (*building block hypothesis* [Holland 75]). Crossover tends to preserve the genetic information present in the parent chromosomes. 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).

After crossover, chromosomes are subjected to random mutations. During mutations, elements of a chromosome are randomly altered. Just as p_{cros} controls the probability of crossover, the mutation rate p_{muta} gives the probability for a given element to be mutated. Mutation is usually treated 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 p_{muta} is usually very small.

Chapter 3

Multiple-Valued Multiple-Threshold Perceptrons

3.1 Introduction

Our main research problem is to synthesize multiple-valued logic functions by neural networks. Most of our research focuses on the study of the computational and learning abilities of multiple-valued multiple-threshold perceptrons. As far as we know, this is a new model for neural networks. Some special cases of this model are known in literature [Obradović 92, Obradović 96], and [Obradović 94] describe learning algorithms for these cases. Our main contribution in the domain is the development of learning algorithms for multiple-valued multiple-threshold perceptrons and the constructions of neural networks composed of such perceptrons. Concepts which are known for the binary perceptron such as linear partition, linear separability and capacity are extended to the multiple-valued perceptron.

3.2 The (n, k, s)-perceptrons

Let $V \subset R^n$ and $K = \{0, \ldots, k-1\}$, with $|V| = v$ and $k \geq 2$. A *n-input k-valued real function* $f : V \mapsto K$ is a function with real-valued inputs and k-valued outputs. When $V \subseteq K^n$, we will refer to f as a *n-input k-valued logic function*. We denote by P_k^n the set of all n-input k-valued real (resp. logic) functions $f : V \mapsto K$ and by $P_k = \bigcup_{n \geq 1} P_k^n$ the set of all k-valued real

(logic) functions. For instance, for $k = 2$, P_2 is the set of all binary logic functions.

In the theory of multiple-valued logic functions there exists an important class of functions called *multiple-valued multiple-threshold functions* (or MVMT functions, for short) [Abd-El-Barr 86b, Haring 65, Ishizuka 77]. Such functions are used in the design of classes of multiple-valued logic circuits called *programmable logic arrays* [Sasao 89].

A *k-valued s-threshold function* of one variable $y \in R$ [Ishizuka 76] is defined as

$$g_{k,s}^{\vec{t},\vec{o}}(y) = \begin{cases} o_0 & \text{if } y < t_1 \\ o_i & \text{if } t_i \leq y < t_{i+1} \text{ for } 1 \leq i \leq s-1 \\ o_s & \text{if } t_s \leq y \end{cases} \tag{3.1}$$

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

Multiple-threshold devices [Haring 65] are thus threshold elements containing multiple levels of excitation (thresholds). They have drawn less enthousiasm in the neural network community. Among their qualities, though, is that given enough thresholds, a single multiple-threshold element can realize any given function operating on a finite domain. For instance, 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 $y = \vec{w}\vec{x} = \sum_{i=1}^{n} w_i x_i$ is called the *excitation* and $\vec{w} = (w_1, \ldots, w_n) \in R^n$ is a weight vector associated with $\vec{x}$ [Abd-El-Barr 86b, Ishizuka 77].

A n-input k-valued s-threshold perceptron [Ngom 98c], abbreviated as *(n, k, s)-perceptron*, computes a *n-input k-valued weighted s-threshold function* $F_{k,s}^{n}(\vec{w}, \vec{t}, \vec{o})$ given by

$$F_{k,s}^{n}(\vec{w}, \vec{t}, \vec{o})(\vec{x}) = g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x}) = \begin{cases} o_0 & \text{if } \vec{w}\vec{x} < t_1 \\ o_i & \text{if } t_i \leq \vec{w}\vec{x} < t_{i+1} \; 1 \leq i \leq s-1 \\ o_s & \text{if } t_s \leq \vec{w}\vec{x} \end{cases} \tag{3.2}$$

where input vector $\vec{x} = (x_1, \ldots, x_n) \in V$, weight vector $\vec{w} = (w_1, \ldots, w_n) \in V$, threshold vector $\vec{t} = (t_1, \ldots, t_s) \in R^s$ and $t_i \leq t_{i+1}$ $(1 \leq i \leq s-1$ and $1 \leq s \leq k^n - 1$, the number of threshold values), and output vector $\vec{o} = (o_0, \ldots, o_s) \in K^{s+1}$. 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 simulates a k-valued function $f : V \mapsto K$. Depending on domain V we will refer either to real or logic (n, k, s)-perceptrons.

A (n, k, s)-perceptron is *monotone* if $\vec{o}$ is monotone, that is $o_0 \leq \cdots \leq o_s$ or $o_0 \geq \cdots \geq o_s$, otherwise it is *nonmonotone*. For example, the well-known *binary perceptron* (which is a $(n, 2, 1)$-perceptron) studied in [Minsky 69] is monotone. The discrete multiple-valued logic neurons described in section 2.5 correspond to k-valued $(k-1)$-threshold perceptrons, that is the $(n, k, k-1)$-perceptrons in our definition. Also, the neulonets' units in section 2.1 are a class of $(n, 3, 2)$-perceptrons whose $\vec{t} = (-d, +d)$, $d \in R$, and $\vec{o} = (1, 0, 2)$.

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 $e \leq p$. 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 *homogeneous* if $s = k - 1$ and $\vec{o}$ is the identity k-permutation on K, that is $o_i = i$ for $0 \leq i \leq s$. It is *permutably homogeneous* if $s + 1 \leq k$ and $\vec{o}$ is a $(s + 1, k)$-permutation on K, otherwise it is *heteregeneous*.

3.3 Decomposition of (n, k, s)-perceptrons

A (n, k, s)-perceptron can be can decomposed into a two-layer network of s $(n, 2, 1)$-perceptrons (usual linear threshold units) and one linear combiner [Abd-El-Barr 86b, Ishizuka 76]. 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

$$g_s^{\vec{t},\vec{o}}(\vec{w}\vec{x}) = o_0 + \sum_{i=1}^{s} a_i g(\vec{w}\vec{x})_i \text{ and } g(\vec{w}\vec{x})_i = \begin{cases} 0 & \text{if } \vec{w}\vec{x} < t_i \\ 1 & \text{if } \vec{w}\vec{x} \geq t_i \end{cases} \tag{3.3}$$

where $a_i = o_i - o_{i-1}$ is the weight associated with $g(\vec{w}\vec{x})_i$ and $1 \leq i \leq s$. Figure 3.1 shows example of such decomposition. Figure 3.2 shows the depth-two network corresponding to equation (3.3). The linear combiner can be replaced by a $(s + 1, k, k - 1)$-perceptron with weight $\vec{a} = (1, a_1, \ldots a_s)$, threshold $\vec{t} = (1, \ldots k - 1)$, and output $\vec{o} = (0, \ldots, k - 1)$.

Multiple-valued neural networks, in our definition, are thus neural networks composed of (n, k, s)-perceptrons as processing units. It should be

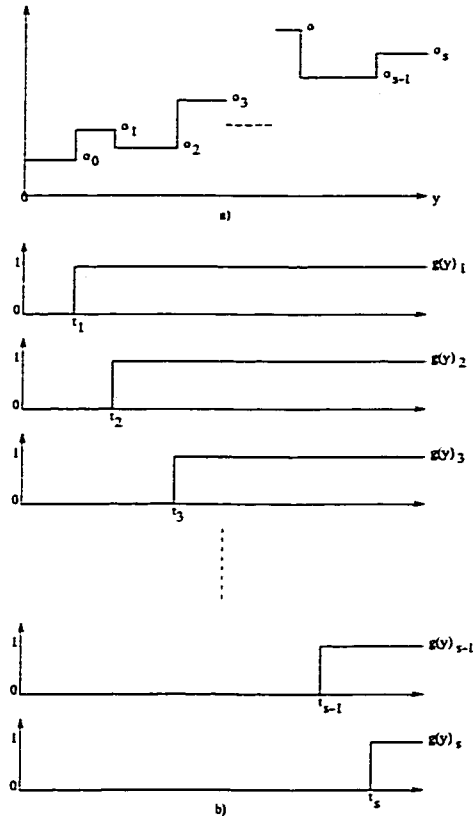


Figure 3.1: Decomposition of $g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x})$ into s linear threshold functions.

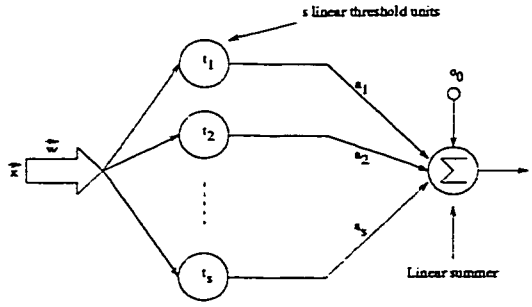


Figure 3.2: Two-layer network for $g_{k,s}^{\vec{t},\vec{o}}(\vec{w}\vec{x})$.

noted that the units are not necessarily the same. For example, one layer of the network may contains $(2, 4, 8)$-perceptrons while another layer may con-

tains $(5, 4, 3)$-perceptrons. The first model of multiple-valued logic neural networks were introduced in [Cao 93] and since then various other models have been described [Ngom 98c, Obradović 92, Tang 95, Watanabe 90].

3.4 Multilinear separability and multilinear partition

The problem of computing (or simulating) a given function f by a (n, k, s)-perceptron, is to determine s and a vector $\vec{r} = (\vec{w}, \vec{t}, \vec{o}) \in R^{n+s} \times K^{s+1}$ such that $F^n_{k,s}(\vec{r})(\vec{x}) = f(\vec{x})$ $(\forall \vec{x} \in V)$, i.e. $f = F^n_{k,s}(\vec{r})$. We will refer to $\vec{r}$ as a *s-representation* of $F^n_{k,s}$ for f.

Let $V = \{\vec{x}_1, \ldots, \vec{x}_v\} \subseteq R^n$ be a set of v vectors $(v \geq 1)$. A k-valued function f with domain V and specified by the input-output pairs $\{(\vec{x}_1, f(\vec{x}_1)), \ldots, (\vec{x}_v, f(\vec{x}_v))\}$, where $\vec{x}_i \in R^n$, $f(\vec{x}_i) \in K$, 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

$$f(\vec{x}_i) = \begin{cases} o_0 & \text{if } \vec{w}\vec{x}_i < t_1 \\ o_j & \text{if } t_j \leq \vec{w}\vec{x}_i < t_{j+1} \text{ for } 1 \leq j \leq s-1 \\ o_s & \text{if } t_s \leq \vec{w}\vec{x}_i \end{cases} \tag{3.4}$$

for $1 \leq i \leq v$. 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 function 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 $V \subset R^n$ into $s+1$ distinct classes $H_0^{[o_0]}, \ldots, H_s^{[o_s]}$, using s parallel hyperplanes, where $H_j^{[o_j]} = \{\vec{x} \in V | f(\vec{x}) = o_j \text{ and } t_j \leq \vec{w}\vec{x} < t_{j+1}\}$. We assume that $t_0 = -\infty$ and $t_{s+1} = +\infty$. Each hyperplane equation denoted by H_j $(1 \leq j \leq s)$ is of the form

$$H_j : \vec{w}\vec{x} = t_j \tag{3.5}$$

Multilinear separability (s-separability) extends the concept of *linear separability* (1-separability of the common binary 1-threshold perceptron) to the (n, k, s)-perceptron. Linear separability in two-valued case tells us that a $(n, 2, 1)$-perceptron can only learn from a space $V \subseteq [0, 1]^n$ in which there is a single hyperplane which separates it into two disjoint halfspaces: $H_0^{[0]} = \{\vec{x} | f(\vec{x}) = 0\}$ and $H_1^{[1]} = \{\vec{x} | f(\vec{x}) = 1\}$. From the *$(n, 2, 1)$-perceptron convergence theorem* [Minsky 69], concepts which are linearly nonseparable cannot be learned by a $(n, 2, 1)$-perceptron. One example of linearly nonseparable two-valued logic function is the n-input parity function. Likewise, the

(n, k, s)-*perceptron convergence theorems* [Ngom 98c, Obradović 94] state that a (n, k, s)-perceptron computes a given function $f \in P_k^n$ if and only if f is s-separable. Figure 3.3 shows an example of 2-separable 4-valued logic function of P_5^2.

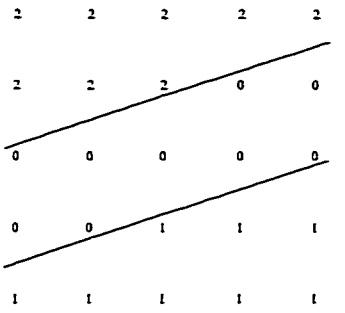


Figure 3.3: A 2-separable function of P_5^2.

A function implementable by a permutably homogeneous (n, k, s)-perceptron is said to be *homogeneously separable* (or homogeneous, for short). Recall that a homogeneous (n, k, s)-perceptron has $s = k-1$ thresholds and that its output vector is the identity $(k-1)$-permutation. A function implementable by a (n, k, s)-perceptron with given output vector $\vec{o}$ is said to be $\vec{o}$-*separable*. A function computable by a monotone (n, k, s)-perceptron is said to be *monotoneously separable*, otherwise it is *nonmonotoneoulsy separable*. For instance, the function in Figure 3.6 is permutably homogeneous, $(3, 1, 2, 0)$-separable, and nonmonotone. Each homogeneously separable function is also monotoneously separable, but not vice versa.

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 one-separable two-valued logic function is (0, 1)-separable, and also, every (0, 1)-separable two-valued logic function is one-separable.

Let $V \subset R^n$ with $|V| = v \geq 2$. A (n, v, s)-*partition* is a partition of V by $s \leq v-1$ parallel hyperplanes (namely $(n-1)$-planes) which do not pass through any of the v points. For instance Figure 3.4 shows an example of $(2, 5^2, 2)$-partition.

A (n, v, s)-partition determines $s+1$ distinct classes $S_0, \ldots, S_s \subset V$ separated by s parallel $(n-1)$-planes such that $\bigcup_{i=0}^{s} S_i = V$ and $\bigcap_{i=0}^{s} S_i = \emptyset$. A (n, v, s)-partition corresponds to a s-separable k-valued function $f \in P_k^n$

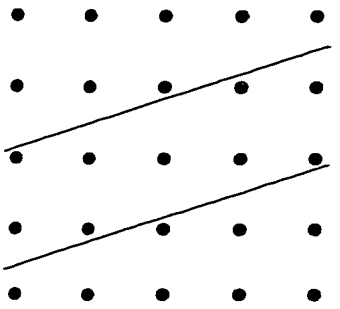


Figure 3.4: A $(2, 5^2, 2)$-partition.

if and only if all points in the set S_i, for $0 \leq i \leq s+1$, have the same value taken out of K. Also, we assume that any two neighboring classes have distinct values.

3.5 The (n, k, s)-perceptron learning problem

Let $f \in P_k^n$ be a target function to learn. The *(n, k, s)-perceptron learning problem* is to search for a s-representation $\vec{r} = (\vec{w}, \vec{t}, \vec{o}) \in R^{n+s} \times K^{s+1}$ such that $F_{k,s}^n(\vec{r})(\vec{x}) = f(\vec{x})$ $(\forall \vec{x} \in V)$, i.e. $f = F_{k,s}^n(\vec{r})$. When s is not given (or not known), then the problem becomes more difficult and the question is how to obtain a *minimal* s-representation $\vec{r}$ for f such that s is as small as possible. We will refer to this problem as the *s-representation problem*

For a fixed s, a threshold vector $\vec{t}$ is *canonical* if for every k-valued logic function f, computable by a (n, 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 (n, k, s)-perceptron computable function f has a s-representation of the form $(\vec{w}, \vec{t}, \vec{o})$, for some $\vec{w}$ and $\vec{o}$. For instance, the vector $\vec{t} = (0)$ is canonical for a $(n, 2, 1)$-perceptron and the vector $\vec{t} = (0, 1)$ is canonical for a $(n, 3, 2)$-perceptron. It was shown in [Obradović 92] that, for homogeneous $(n, k, k-1)$-perceptrons, there is no canonical threshold vector $\vec{t} \in R^{k-1}$ when $k \geq 4$. This suggests that, for such perceptrons, $\vec{t}$ must be learned in addition to the weight vector $\vec{w}$.

[Obradović 94] proposed a learning algorithm for the homogeneous $(n, k, k-1)$-perceptrons (we call it *homogeneous $(n, k, k-1)$-perceptron learning algorithm*). For a homogeneous $(n, k, k-1)$-perceptron the output vector is always set to $\vec{o} = (0, \ldots, k-1)$ (that is, $o_i = i$ for $0 \leq i \leq k-1$), and its learning algorithm is shown in Figure 3.5.

```
Procedure MultiPerceptron(f, n, k, o⃗ = (0, ..., k − 1));
    w⃗ := 0⃗;
    t⃗ := 0⃗;
    Repeat
        for each x⃗ ∈ K^n do
            v := F^n_{k,k−1}(w⃗, t⃗, o⃗)(x⃗);
            if f(x⃗) ≠ v then
                MultiPerceptronUpdateRule(x⃗, f(x⃗), v);
        Output (w⃗, t⃗);
    Until F^n_{k,k−1}(w⃗, t⃗, o⃗) = f;

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

Figure 3.5: Homogeneous $(n, k, k-1)$-perceptron learning algorithm.

In Figure 3.5, the constant $0 < \eta \leq 1$ is the learning rate. The initial weights can be set to any (random) values. The initial thresholds can be set to a (random) constant, 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. $c = kn$). We must ensure that $t_{v-1} \leq t_v \leq t_{v+1}$ each time we update t_v or t_{v+1}). We can also generate a new random η before each call to *MultiPerceptronUpdateRule*.

In the algorithm, the weight and threshold vectors are always updated in opposite directions using the error value $\delta = f(\vec{x}) - v$. If $f(\vec{x}) < v$ then $\delta < 0$ means that the weights are too large or t_v is too small. Therefore we decrease the weights and increase t_v. If $f(\vec{x}) > v$ then $\delta > 0$ means that the weights are too small or t_{v+1} is too large. Thus we increase the weights and decrease t_{v+1}. When $f(\vec{x}) = v$ no modification is done and the algorithm goes to the next step.

As a consequence of the binary perceptron convergence theorem [Duda 73, Minsky 69, Nilsson 62, Novikoff 62], it is proven in [Obradović 94] that the

homogeneous $(n, k, k-1)$-perceptron learning algorithm terminates if and only if there exists a $(k-1)$-representation $(\vec{w}, \vec{t}, \vec{o})$ for f.

For a fixed number of thresholds s, the problem of learning a (n, 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 learned along with the vectors $\vec{w}$ and $\vec{t}$. For instance, the function shown in Figure 3.6 cannot be computed (hence cannot be learned) by a homogeneous $(n, 4, 3)$-perceptron. This function can be learned (hence computed) by a nonmonotone $(n, 4, 3)$-perceptron with $g_3^{\vec{t},\vec{o}=(0,2,1,3)}$ as nonmonotone transfer function. A monotone (n, k, s)-perceptron cannot simulate a k-valued logic function whose space distribution is nonmonotone.

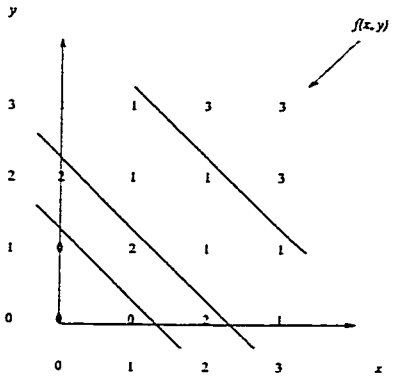


Figure 3.6: Example of three-separable two-input four-valued logic function.

Chapter 4

On the Computing Capacity of Multiple-Valued Multiple-Threshold Perceptrons

In this chapter, we propose to explore the computational power of (n, k, s)-perceptrons. More specifically, the question we will attempt to answer is the following. How many k-valued functions $f : V \mapsto K$ can be simulated by a (n, k, s)-perceptron? This will be refered to as the *capacity* of a (n, k, s)-perceptron. Based on well known relationships between linear partitions and minimal pairs, we derive formulae for the number of multilinear partitions of a point set in general position and of the set K^2. The (n, k, s)-perceptrons partition the input space V into $s+1$ regions with s parallel hyperplanes. We obtain results on the capacity of a single (n, k, s)-perceptron, respectively for $V \subset R^n$ in general position and for $V = K^2$. Finally, we describe a fast polynomial-time algorithm for counting the multilinear partitions of K^2.

4.1 Introduction

There has been an intensive interest in threshold logic as the main component of neural network models. These models provide a direction for pattern recognition systems with distinct natural advantages. The capacity of these models, as well as their computing power, are directly related to the number of threshold functions.

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 network models based on threshold logic. It is therefore of practical as well as theoretical interest to estimate the number of functions that can be modeled as multiple-threshold functions for a given number of inputs and threshold levels.

Our question is important from the point of view of computational learning theory. If we know the answer then it may be possible to obtain results on the Vapnik-Chervonenkis dimension of (n, k, s)-perceptrons. As a consequence, the (n, k, s)-perceptrons may be PAC-learnable. That is, there may exists an efficient (n, k, s)-perceptron learning algorithm which uses only a polynomial number of examples and that, with high probability (which can be made as high as desired), the algorithm outputs a good approximation of a given function within a desired degree of accuracy (which can be made as high as desired) [Anthony 92].

The number of k-valued functions that can be defined over V is k^v. The first question we address is: how many of these k^v functions can be computed by a single (n, k, s)-perceptron with given output vector $\vec{o}$? In other words, given an output vector $\vec{o}$, how many k-valued functions defined over V are $\vec{o}$-separable? The second question we address is: for a fixed s, how many k-valued functions over V are s-separable? Or equivalently, how many (n, k, s)-perceptrons can be defined over V?

Unlike the $(n, 2, 1)$-perceptron, for $k \geq 3$ a (n, k, s)-perceptron learns only a very small subset of the set of s-separable k-valued logic functions. Consider for instance the 2-input 4-valued functions in Figure 3.6. Clearly this function is three-separable since we can draw three lines that partition the inputs into four disjoint classes.

A homogeneous $(n, 4, 3)$-perceptron (for instance) will learn $(0, 1, 2, 3)$-separable function f_1 and not be able to learn $(0, 2, 1, 3)$-separable function f_2 (even though both functions are 3-separable). The reason for this is that there is no output vector $\vec{o} \in K^{s+1}$ such that both f_1 and f_2 are $\vec{o}$-separable. Clearly, a monotone (n, k, s)-perceptron learning algorithm will never terminate in learning a nonmonotoneously separable function.

If for a given (n, v, s)-partition we have $S_i \neq \emptyset$ $(0 \leq i \leq s + 1)$ then, clearly, the number of associated functions is $k(k-1)^s$. In this section, we consider only partitions where $S_i \neq \emptyset$ for $0 \leq i \leq s$.

For given $s \geq 2$, the capacity of a (n, k, s)-perceptron with domain V is *approximated* by the product of the number of (n, v, s)-partitions and the number of functions associated with each (n, v, s)-partition of V. Thus counting the number of (n, v, s)-partitions of V is a first step toward calcu-

lating the capacity of (n, k, s)-perceptrons. We emphasize that the s partitioning $(n-1)$-planes of a (n, v, s)-partition do not pass through any point of V, and therefore we do not obtain the exact number of (n, k, s)-perceptron computable functions.

A *linear partition* of a point set V is a $(n, v, 1)$-partition, so only a single $(n-1)$-plane is required to separate a n-dimensional space $V \subseteq R^n$ into two halfspaces. The enumeration problem for linear partitions is closely related to the efficiency measurement problem for linear discriminant functions in pattern recognition [Duda 73] and to many other algorithmic problems [Edelsbrunner 87].

The capacity of $(n, 2, 1)$-perceptrons with domain V [Siu 95] is well-known and is given by

$$|F^n_{2,s}| = 2\sum_{i=0}^{n}\binom{v-1}{i} = \begin{cases} 2^v & \text{if } n \geq v-1 \\ 2^v - 2\sum_{i=n+1}^{v-1}\binom{v-1}{i} & \text{otherwise} \end{cases} \tag{4.1}$$

[Olafsson 88] estimated lower and upper bounds for the capacity of $(n, 2, s)$-perceptrons, using two essentially different enumeration techniques. The paper demonstrated that the exact number of multiple-threshold functions depends strongly on the relative topology of the input set. The results corrected a previously published estimate [Takiyama 85] and indicated that adding threshold levels enhances the capacity more than adding variables.

4.2 Linear partitions and minimal pairs

An unordered pair $(\vec{x}, \vec{y})$ of distinct points of a finite set $V \subseteq R^n$ is a *minimal pair* (with respect to V) if there does not exist a third point $\vec{z}$ of V which belongs to the open line segment $[\vec{x}, \vec{y}]$.

A point set $V \subset R^n$ is said to be in *general position* if and only if no subset of V of $d+1$ points lies on a $(d-1)$-plane (for $1 \leq d \leq n$) and no two $(d-1)$-planes defined from V are parallel (for $2 \leq d \leq n$). The number of linear partitions of a finite planar point set, no three points of which are collinear, is well known [Tou 74]. The number of linear partitions of V in general position is also a well known result of pattern recognition [Cover 65, Nilsson 68], and is given by

$$L_{n,v,1} = \sum_{i=0}^{n}\binom{v-1}{i} \tag{4.2}$$

A relationship between linear partitions and minimal pairs of a finite point set V in the plane was established in [Koplowitz 88]. Namely, that the number of linear partitions of V is equal to the number of minimal pairs in V. Moreover, each minimal pair determines two linear partitions of V (see Figure 4.1) and conversely, each linear partition of V has exactly two associated minimal pairs (see Figure 4.2) [Acketa 91]. These facts provide an easier way of counting linear partitions for arbitrary finite planar point sets: it suffices to count minimal pairs.

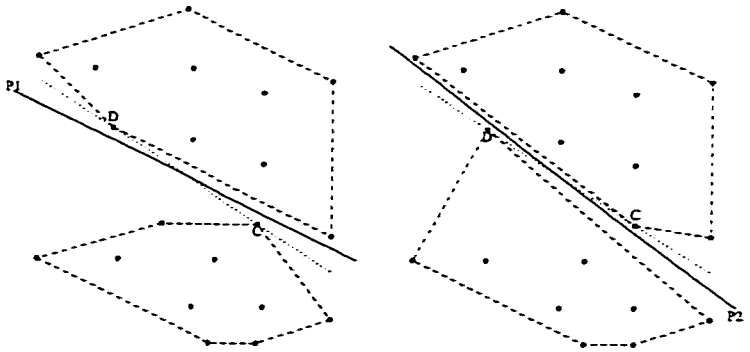


Figure 4.1: Minimal pair (C, D) corresponds to separating lines P_1 and P_2.

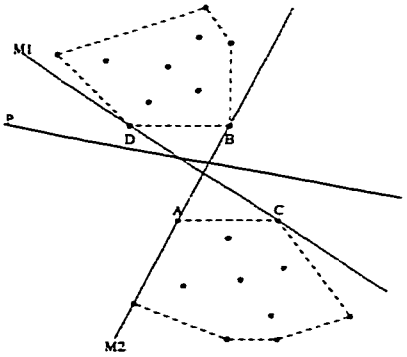


Figure 4.2: Separating line P corresponds to minimal pairs (A, B) and (C, D).

In [Acketa 91] an explicit formula for the number of linear partitions of the (i, j)-grid (a rectangular part of the infinite grid) is stated, using the correspondence with minimal pairs. The same reference includes an efficient algorithm for counting linear partitions of the (i, j)-grid in linear time with respect to the number of its points.

[Tošić 92] derived a formula for the number of linear partitions of a given point set V in two-dimensional and three-dimensional spaces, depending on the configuration formed by the points of V. He considered the case where some points of V may coincide.

4.3 Counting multilinear partitions of subsets in general position

In this section, $V \subset R^n$ is in general position and $L_{n,v,s}$ denotes its number of (n, v, s)-partitions. Clearly, every pair of points in V is a minimal pair. We first obtain the number of $(2, v, s)$-partitions, that is for the two dimensional space, and then we find $L_{n,v,s}$.

We rewrite the number of linear partitions of V in general position [Cover 65, Nilsson 68] with a little modification as

$$L_{n,v,1} = L_{n,v-1,1} + L_{n-1,v-1,1} = \sum_{i=1}^{n} \binom{v-1}{i} \tag{4.3}$$

The difference with the original formula (4.2) is that the index i starts with 1 instead of 0. The reason we start from $i = 1$ is that we do not include partitions containing empty classes.

To count the $(2, v, s)$-partitions, we associate each $(2, v, s)$-partition with a slope $\sigma \in R$ (or equivalently, a minimal pair) as follows.

- For each of the s partitioning lines, choose the corresponding minimal pair which has the smaller slope;
- The associated slope of the given $(2, v, s)$-partition is the maximum among these smaller slopes.

Lemma 4.3.1 *The number of $(2, v, s)$-partitions associated with a given slope σ is* $\binom{v-2}{s-1}$.

Proof For a given minimal pair, rotate its slope σ to increase it a bit in order to obtain the direction of separation P that corresponds to σ. Sort the points of V along this direction, that is according to their distance to the line P. Consider the selected minimal pair as one point. Then we can choose $s - 1$ additional separating lines (parallel to P) for $v - 2$ points in $\binom{v-2}{s-1}$ ways. •

Theorem 4.3.1 $L_{2,v,s} \leq \binom{v-2}{s-1}\binom{v}{2}$.

Proof Since V is in general position, then there are $\binom{v}{2}$ slopes. The inequality is explained by the fact that two distinct choices of minimal pairs (or slopes) may have sets of associated $(2, v, s)$-partitions that intersect each other. Therefore a given partition may be counted many times, depending on the configuration of V. For instance consider one of the partitions in Figure 4.3. Clearly, the same partition can be obtained either by selecting the upper minimal pair or by selecting the lower minimal pair (indeed in this example, they both give the same set of associated partitions, even though the points are in general position). We have equality only when $s = 1$. •

Theorem 4.3.2

$$L_{n,v,s} \leq \binom{v-2}{s-1} L_{n,v,1} = \binom{v-2}{s-1} \sum_{i=1}^{n} \binom{v-1}{i}.$$

Proof The proof is similar to that of Theorem 4.3.1. We select a linear partition of V in $L_{n,v,1}$ ways. For each such linear partition, we sort the points of V along the direction of the separating $(n-1)$-plane, that is according to their distance to that hyperplane. To construct a (n, v, s)-partition we must add $s-1$ more parallel $(n-1)$-planes. There are $\binom{v-2}{s-1}$ ways to place $s-1$ new hyperplanes in $v-2$ available positions. Here again we have equality only for $s = 1$. •

Corollary 4.3.1 *The number of n-input k-valued s-separable functions $f : V \mapsto K$ is*

$$|F_{k,s}^{n}| \leq k(k-1)^{s} \binom{v-2}{s-1} \sum_{i=1}^{n} \binom{v-1}{i}.$$

Proof Given a (n, v, s)-partition $S_0, \ldots, S_s$, each class can take one of k values from K such that any two neigbhoring classes have different values. So there are $k(k-1)^s$ ways to assign values to a (n, v, s)-partition. Each assignment of values to a (n, v, s)-partition defines a unique k-valued s-separable function. The inequality comes from the fact that some functions can be obtained by at least two different (n, v, s)-partitions (Figure 4.3 shows an example of such function). We have equality only when $s = 1$. •

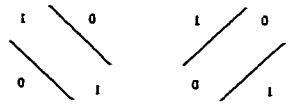


Figure 4.3: Two $(2, 4, 2)$-partitions for the same function.

For $k = 2$, Corollary 4.3.1 gives a much better (tighter) upper bound on the capacity of $(n, 2, s)$-perceptrons than the following upper bound obtained in [Olafsson 88].

$$|F_{2,s}^n| \leq 2 \left(\begin{array}{c} v-1 \\ s \end{array} \right) \sum_{i=0}^{n-1} \left(\left(\begin{array}{c} v \\ 2 \end{array} \right) - 1 \right) \qquad (4.4)$$

However, in (4.4), the constant $\left(\begin{array}{c} v-1 \\ s \end{array} \right)$ also counts partitions containing empty classes. So, to remove them, we must replace it by $\left(\begin{array}{c} v-2 \\ s-1 \end{array} \right)$.

A s-separable function $f : V \mapsto K$ is *permutably homogeneous* if and only if $s \leq k-1$ and, for any partitioning of V, no two separate classes have equal value [Ngom 98c]. These functions are determined by (n, v, s)-partitions in which each of the $s+1$ classes $S_0, \ldots, S_s$ maps to a distinct value in K.

Theorem 4.3.3 *A permutably homogeneous s-separable function $f : V \mapsto K$ has a unique (n, v, s)-partition.*

Proof Suppose f has two distinct (n, v, s)-partitions P_1 and P_2. Consider two points $\vec{x}$ and $\vec{y}$ which are in the same class with respect to P_1 but in different classes with respect to P_2. We have that $f(\vec{x}) = f(\vec{y})$, because $\vec{x}$ and $\vec{y}$ share the same class in P_1. This means that two distinct classes in P_2 have the same value. This contradicts the definition of permutably homogeneous function. •

Corollary 4.3.2 *The number of permutably homogeneous n-input k-valued s-separable functions $f : V \mapsto K$ is*

$$|G_{k,s}^n| \leq \frac{k!}{(k-s-1)!} \left(\begin{array}{c} v-2 \\ s-1 \end{array} \right) \sum_{i=1}^{n} \left(\begin{array}{c} v-1 \\ i \end{array} \right).$$

Proof For a (n, v, s)-partition where $s \leq k-1$, the number of ways to map the $s+1$ classes $S_0, \ldots, S_s$ to distinct values in K equals the number of $(s+1, k)$-permutations, that is $\frac{k!}{(k-s-1)!}$. Each such $(s+1, k)$-permutation uniquely determines a permutably homogeneous s-separable function. From Theorem 4.3.3, each permutably homogeneous s-separable function uniquely determines a (n, v, s)-partition. We have equality when $s = 1$. •

4.4 Counting multilinear partitions of the (k, k)-grid

In this section we generalize the counting method of [Acketa 91] to enumerate multilinear partitions of the (k, k)-grid K^2.

Let $(\vec{x}, \vec{y})$ be a pair of points from a finite planar grid. Let $a = \max(|x_2 - x_1|, |y_2 - y_1|)$ and $b = \min(|x_2 - x_1|, |y_2 - y_1|)$. One well known condition for a pair $(\vec{x}, \vec{y})$ to be minimal is that a and b are relatively prime, that is their greatest common divisor is 1. $a \perp b$ will denote that the integers a and b are mutually simple.

Thus the number of minimal pairs corresponds to the number of pairs $(\vec{x}, \vec{y})$ of the (i, j)-grid such that $a \perp b$. Let natural numbers i and j be given so that $i \leq j$. The *generalized Farey (i, j)-sequence* $F_{i,j}$ [Acketa 91] is the strictly increasing sequence of all the fractions of the form $\frac{b}{a}$, where the integers a and b satisfy: $a \perp b$, $0 < b < a \leq j$, $b \leq i$. Thus the sequence $F_{4,7}$ is as follows:

$$\frac{1}{7}\ \frac{1}{6}\ \frac{1}{5}\ \frac{1}{4}\ \frac{2}{7}\ \frac{1}{3}\ \frac{2}{5}\ \frac{3}{7}\ \frac{1}{2}\ \frac{4}{7}\ \frac{3}{5}\ \frac{2}{3}\ \frac{3}{4}\ \frac{4}{5}.$$

The *Farey i-sequence* F_i for any positive integer i is the set of irreducible rational numbers $\frac{b}{a}$, with $0 \leq b \leq a \leq i$ and $a \perp b$, arranged in increasing order [Hardy 54]. So the sequence F_4 is:

$$\frac{0}{1}\ \frac{1}{4}\ \frac{1}{3}\ \frac{1}{2}\ \frac{2}{3}\ \frac{3}{4}\ \frac{1}{1}.$$

The length of the sequence $F_{i,j}$ (resp. F_i) will be denoted by $|F_{i,j}|$ (resp. $|F_i|$). Also, $F_{i,j}^d$ (resp. F_i^d) stands for the d-th fraction in $F_{i,j}$ (resp. F_i), $1 \leq d \leq |F_{i,j}|$ (resp. $|F_i|$).

To count the $(2, k^2, s)$-partitions, we associate to each $(2, k^2, s)$-partition a fraction $\frac{b}{a} \in F_{k-1}$ as follows:

- For each of the s partitioning lines, choose one of the two minimal pairs which has the smaller slope;

- The associated slope of the given $(2, k^2, s)$-partition is the maximum among these smaller slopes.

Then to enumerate or generate the $(2, k^2, s)$-partitions associated with a given $\frac{b}{a}$ we will need Lemmas 4.4.1 and 4.4.2 below. As in Figure 4.4, rotate a line segment $[\vec{x}, \vec{y}]$ whose slope $\frac{b}{a}$ is irreducible to increase it for a *small* amount, so that we obtain a straight line P with slope $m \notin F_{k-1}$. P is the direction for separation and corresponds to the line segment $[\vec{x}, \vec{y}]$ with slope $\frac{b}{a}$.

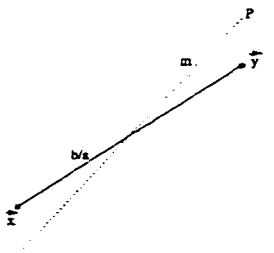


Figure 4.4: Rotating slope $\frac{b}{a}$ toward slope m.

Lemma 4.4.1 *Slope $\frac{b}{a}$ of the line segment $[\vec{x}, \vec{y}]$ is greater than or equal to the associated slope of any $(2, k^2, s)$-partition in direction parallel to P, where P is the direction of separation that corresponds to the line segment $[\vec{x}, \vec{y}]$.*

Proof Consider Figure 4.5, where $t \in F_{k-1}$ is the associated slope of a $(2, k^2, s)$-partition parallel to P and $m \notin F_{k-1}$ is the slope of P. From our construction of P we have that $m > \frac{b}{a} \in F_{k-1}$ but near $\frac{b}{a}$. Clearly, from Figure 4.5, $t < m$. Let d be the rank of $\frac{b}{a}$ in F_{k-1}. Since $F_{k-1}^d = \frac{b}{a} < m < F_{k-1}^{d+1}$, $t < m$ and $t \in F_{k-1}$, therefore we have $t \leq \frac{b}{a}$. •

Lemma 4.4.2 *Slope $\frac{b}{a}$ is equal to the associated slope of a $(2, k^2, s)$-partition in direction parallel to P if and only if at least one of the s separating lines intersects a minimal pair with slope $\frac{b}{a}$.*

Proof Let $t \in F_{k-1}$ be the associated slope of a $(2, k^2, s)$-partition parallel to P. $\Rightarrow$) If none of the separating lines intersects any minimal pair with slope $\frac{b}{a}$ then, $t \neq \frac{b}{a}$. Moreover from Lemma 4.4.1 we obtain $t < \frac{b}{a}$. $\Leftarrow$) If one of the separating lines intersects a minimal pair with slope $\frac{b}{a}$ then,

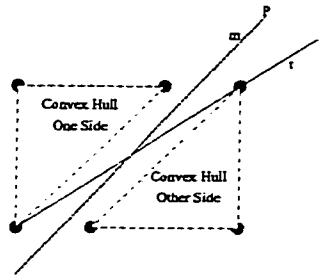


Figure 4.5: $t \leq \frac{b}{a} < m$.

from Lemma 4.4.1 $t = \frac{b}{a}$. •

[Acketa 91] obtained an exact formula for the number of linear partitions of the (i, j)-grid. Substituting for the (k, k)-grid we obtain the following corollary.

Corollary 4.4.1

$$L_{2,k^2,1} = 2k(k-1) + 2(k-1)^2 + 4 \sum_{a \perp b, 0<b<a<k} (k-a)(k-b).$$

Proof If $b = 0$, then the number of minimal pairs is equal to $2k(k-1)$ which is the number of minimal vertical and horizontal segments of the (k, k)-grid (these are $k(k-1)$ each). This argument explains the first sum. If $b > 0$ and $a \perp b$ then $a = b$ implies $a = b = 1$. In that case, the (k, k)-grid contains obviously $2(k-1)^2$ minimal segments with slope $\frac{1}{1}$, which explains the second sum. Finally, if $a > b > 0$ and $a \perp b$, then there are exactly $2(k-a)(k-b)$ horizontal and vertical rectangles of size $a \times b$ in the (k, k)-grid. The third sum comes from the fact that there are two minimal segments per each rectangle (the diagonal segments). •

Lemma 4.4.3

$$\begin{aligned} L_{2,k^2,s} &= 4\binom{k^2-1}{s} - 2\binom{k-1}{s} - 2\binom{2k-2}{s} + \\ &\quad 4 \sum_{a \perp b, 0<b<a<k} \left[\binom{k^2-1}{s} - \binom{ak+bk-ab-1}{s} \right]. \end{aligned}$$

Proof Let a slope $\frac{b}{a}$ be such that $a \perp b$ (we consider slopes for directions between 0° and 45°. For each such slope $\frac{b}{a}$, Lemmas 4.4.1 and 4.4.2

give a simple algorithm to construct (i.e generate) and count the $(2, k^2, s)$-partitions associated with $\frac{b}{a}$.

- Rotate slope $\frac{b}{a}$ to increase it a bit {this gives the direction of an associated $(2, k^2, s)$-partition P};
- Sort the points along this direction;
- Choose s points $\vec{x}_1, \ldots, \vec{x}_s$ out of $k^2 - 1$ points in $\begin{pmatrix} k^2-1 \\ s \end{pmatrix}$ ways; {selected points are begining of classes $S_1, \ldots, S_s$ and the point $\vec{x}_0 = (k-1, 0)$ can always be selected for S_0}
- Eliminate all selections of points where no minimal pair of slope $\frac{b}{a}$ is intersected by a separating line of direction P;

To intersect one of the s separating lines, the lower end of a minimal pair with slope $\frac{b}{a}$ must be selected. From Corollary 4.4.1 we have

- If $b = 0$ then there are $k(k-1)$ lower ends (the number of minimal horizontal segments of the (k, k)-grid). None of them can be selected in $\begin{pmatrix} k^2-1-k(k-1) \\ s \end{pmatrix} = \begin{pmatrix} k-1 \\ s \end{pmatrix}$ ways. So the number of ways to select a minimal pair with slope $\frac{0}{1}$ to intersect a separating line is $2\begin{pmatrix} k^2-1 \\ s \end{pmatrix} - 2\begin{pmatrix} k-1 \\ s \end{pmatrix}$.
- If $a = b = 1$ then there are $(k-1)^2$ lower ends (the number of minimal segments with slope 45°). None of them can be selected in $\begin{pmatrix} k^2-1-(k-1)^2 \\ s \end{pmatrix} = \begin{pmatrix} 2k-2 \\ s \end{pmatrix}$ ways. So the number of ways to select a minimal pair with slope $\frac{1}{1}$ to intersect a separating line is $2\begin{pmatrix} k^2-1 \\ s \end{pmatrix} - 2\begin{pmatrix} 2k-2 \\ s \end{pmatrix}$.
- If $a > b > 0$ then there are $(k-a)(k-b)$ lower ends (the number of horizontal rectangles of size $a \times b$ of the (k, k)-grid). None of them can be selected in $\begin{pmatrix} k^2-1-(k-a)(k-b) \\ s \end{pmatrix} = \begin{pmatrix} ak+bk-ab-1 \\ s \end{pmatrix}$ ways. So the number of ways to select a minimal pair with slope

$0 < \frac{b}{a} < 1$ to intersect a separating line is
$4 \sum_{a \perp b, 0<b<a<k} \left[\binom{k^2-1}{s} - \binom{ak+bk-ab-1}{s} \right]$.

Taking the total sum of all three cases yields the formula. This completes the proof. •

Theorem 4.4.1

$$L_{2,k^2,s} = 4\binom{k^2-1}{s}|F_{k-1,k-1}| - 4\binom{k^2-1}{s} - 2\binom{k-1}{s} - 2\binom{2k-2}{s} - 4\sum_{a \perp b, 0<b<a<k} \binom{ak+bk-ab-1}{s}.$$

Proof Follows from Lemma 4.4.3 and the fact that the last sum is over $|F_{k-1,k-1}|$. •

Lemma 4.4.4 $ak + bk - ab - 1 < k^2 - 1$.

Proof Lemma follows from $b < k$ and $a < k$. •

Corollaries 4.4.2 and 4.4.3 below give, respectively, a lower bound and an upper bound on the number of $(2, k^2, s)$-partitions of the (k, k)-grid.

Corollary 4.4.2

$$L_{2,k^2,s} > 4\binom{k^2-1}{s}|F_{k-1,k-1}| - 2\binom{k-1}{s} - 2\binom{2k-2}{s}.$$

Proof Follows from Lemma 4.4.4 and Theorem 4.4.1 because the last sum becomes smaller when substituting $k^2 - 1$ for $ak + bk - ab - 1$. •

Corollary 4.4.3

$$L_{2,k^2,s} < 4\binom{k^2-1}{s}(|F_{k-1,k-1}| + 1) - 2\binom{k-1}{s} - 2\binom{2k-2}{s}.$$

Proof Follows from Theorem 4.4.1 because the last sum is added. •

The asymptotic formula for the length of the generalized *Farey* (i, j)-sequence (for $i \leq j$) is given in [Žunić 91] as $|F_{i,j}| = \frac{3i^2j^2}{\pi^2} + O(i^2 j \log j) + O(ij^2 \log\log j)$. Hence, substituting for the (k, k)-grid we obtain $|F_{k-1,k-1}| = \frac{3k^4}{\pi^2} + O(k^3 \log k) + O(k^3 \log\log k)$.

Theorem 4.4.2

$$L_{2,k^2,s} \approx 4\binom{k^2-1}{s}\frac{3k^4}{\pi^2} - 2\binom{k-1}{s} - 2\binom{2k-2}{s} + 2\binom{k^2-1}{s}.$$

Proof We take the mean of the lower and upper bounds and replace $|F_{k-1,k-1}|$ by its formula given above. •

Corollary 4.4.4 *The number of 2-input k-valued s-separable logic functions is* $|F^2_{k,s}| < k(k-1)^s L_{2,k^2,s}$.

Proof See Corollary 4.3.1. •

The example given in Figure 4.3 explains the inequality in Corollary 4.4.4. For instance, the XOR funcion has exactly two $(2,4,2)$-partitions. For $s = 1$ we obtain an asymptotic formula.

Corollary 4.4.5 *The number of permutably homogeneous 2-input k-valued s-separable logic functions is* $|G^2_{k,s}| \approx \frac{k!}{(k-s-1)!} L_{2,k^2,s}$.

Proof See Corollary 4.3.2. •

4.5 Complexity of counting the number of $(2, k^2, s)$-partitions

Given a point set V in the plane, there are $\frac{v^2-v}{2}$ pairs of points. Generally speaking, each pair requires $v-2$ tests to check whether it is minimal. Thus the complexity of the general case for the extraction of all minimal pairs is $O(v^3)$. However, if the considered $v = ij$ points are all the points of the (i,j)-grid, then this algorithm turns out to be linear with respect to v, namely $O(v)$, on the basis of Corollary 4.4.1 and the fact that the successor of a member of $F_{i,j}$ can be calculated in a constant time [Acketa 91]. The simple linear time algorithm for generating $F_{i,j}$ is described in [Acketa 91].

In Figure 4.5 we show the fast algorithm for counting $(2, k^2, s)$-partitions of the (k,k)-grid. It is a modified version of the [Acketa 91] algorithm for enumerating the linear partitions of the (i,j)-grid, so, the reader is refeered to [Acketa 91] for a complete proof of correctness of the algorithm . The modifications include replacing the (i,j)-grid by the (k,k)-grid and computing the binomial coefficients present in Lemma 4.4.3.

Let $\frac{j}{i}$ and $\frac{d}{c}$ be two consecutive elements of $F_{k-1,k-1}$. It has been shown in [Acketa 91] that the immediate successor $\frac{b}{a}$ of $\frac{d}{c}$ is always determined by

Read k and s;
$d := 1; c := k-1$;
$b := 1; a := k-2$;
$L_{2,k^2,s} := 12\binom{k^2-1}{s} - 2\binom{k-1}{s} - 2\binom{2k-2}{s} - 4\binom{k^2-k}{s} - 4\binom{k^2-2k+1}{s}$;
Repeat
  $j := d; i := c$;
  $d := b; c := a$;
  $r := \left\lfloor \frac{k-1+i}{c} \right\rfloor$;
  $b := rd - j$;
  $a := rc - i$;
  $L_{2,k^2,s} := L_{2,k^2,s} + 4\binom{k^2-1}{s} - 4\binom{ak+bk-ab-1}{s}$;
Until $b = k-2$ and $a = k-1$;
Write out $L_{2,k^2,s}$;

Figure 4.6: Fast algorithm for counting $(2, k^2, s)$-partitions.

the relations $b = rd - j$ and $a = rc - i$, where $r = \left\lfloor \frac{k-1+i}{c} \right\rfloor$. This implies that the computation of a consecutive member of $F_{k-1,k-1}$ can be completed in a constant time. Another consequence of this fact is that initialization is easy: we can always start with the first two *Farey* numbers $F^1_{k-1,k-1} = \frac{1}{k-1}$ and $F^2_{k-1,k-1} = \frac{1}{k-2}$ and then generate the whole $F_{k-1,k-1}$ sequence in a loop using the above relations.

The last sum of the formula given in Lemma 4.4.3 necessarily reduces to the sum over the *Farey* $(k-1, k-1)$-sequence. Suppose that k^2 and s require each $O(\log k^2)$ and $O(\log s)$ bits for memory storage, then the complexity of computing the binomial coefficients is $O(s)$. Therefore, for each generation of a number $\frac{b}{a} \in F_{k-1,k-1}$, it takes $O(s)$ to compute the number of $(2, k^2, s)$-partitions associated with $\frac{b}{a}$. Since there are k^2 elements in the (k, k)-grid, then it takes $O(sk^2) \leq O(k^4)$ time to calculate $L_{2,k^2,s}$ (recall that $s \leq k^2 - 1$). Clearly, the time complexity is polynomial on k.

4.6 Conclusion and further research

In this chapter we have derived formulae for the number of multilinear partitions of subsets in general position and of the (k, k)-grids. The counting techniques we used for the (k, k)-grid apply as well for the (i, j)-grid, where $i \leq j$. A more difficult problem would be to enumerate partitions for higher dimension grids.

An important complexity measure of neural networks is the *Vapnik-Chervonenkis dimension.* It is defined as the maximum size of a training set $T \subseteq V$ such that a given network realizes all functions defined on T. If the (n, k, s)-perceptrons have a finite VC-dimension then they may be PAC-learnable. That is, there may exist an efficient (n, k, s)-perceptron learning algorithm which uses only a polynomial number of examples and that, with high probability (which can be made as high as desired), the algorithm outputs a good approximation of a given function within a desired degree of accuracy (which can be made as high as desired). The VC-dimension of (n, k, s)-perceptrons gives a valid lower bound on the VC-dimension of linear decision lists and other related learning architectures.

One interesting open question is the following. In how many ways can we partition a finite set $V \subset R^n$ using s hyperplanes? The hyperplanes are not necessarily parallel. The answer to this question gives (bounds on) the capacity of neural networks constructed by partitioning algorithms. Such neural network architectures include neural trees and neural decision lists.

Chapter 5

Learning with Permutably Homogeneous Multiple-Valued Multiple-Threshold Perceptrons

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 thesis. 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.

5.1 Introduction

The problem we address in this chapter is that of learning multiple-valued functions by (n, k, s)-perceptrons.

[Obradović 94, Obradović 90a] proposed a learning algorithm for homogeneous $(n, k, k-1)$-perceptrons. As a consequence of the (n,2, 1)-perceptron convergence theorem [Duda 73, Minsky 69, Nilsson 62, Novikoff 62], it is proven in [Obradović 94] 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 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-value functions (for a fixed s). In the next section we propose a (n, k, s)-perceptron learning algorithm that learns a larger class of functions.

For a fixed number of thresholds s, the problem of learning a (n, 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, the function in Figure 3.6 cannot be computed (hence cannot be learned) by a homogeneous $(n, 4, 3)$-perceptron. This function can be learned (hence computed) by a permutably homogeneous $(n, 4, 3)$-perceptron with $g_{4,3}^{\vec{t},\vec{o}=(3,1,2,0)}$ as nonmonotone transfer function. A monotone (n, k, s)-perceptron cannot simulate a nonmonotoneously separable k-valued function.

5.2 Permutably homogeneous (n, k, s)-perceptrons

In this section we propose a learning algorithm for permutably homogeneous (n, 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 [Obradović 94, Obradović 90a] in that they can learn a larger class of multiple-valued functions.

5.2.1 Permutably homogeneous (k, s)-perceptron learning algorithm

A permutation does not need to contain all values of K. There are k-valued functions whose set of output values is a subset $S \subseteq K$, that is functions of the form $f : V \mapsto S \subseteq K$. Examples of such functions are, for instance, functions of the form $f : V \mapsto \{0, 1\}$, that is functions with 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 5.1.

```
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 5.1: Permutably homogeneous (n, k, s)-perceptron learning algorithm.

In Figure 5.1, the constant $0 < \eta \leq 1$ 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 $t_{v-1} \leq t_v \leq t_{v+1}$ each time we update t_v or t_{v+1}. We can also generate a new random η 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 $\delta < 0$ 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 $Pos_{\vec{\sigma}}[f(\vec{x})] > Pos_{\vec{\sigma}}[v]$ then $\delta > 0$ means that the weights are too small or $t_{Pos_{\vec{\sigma}}[v]+1}$ is too large. Thus we increase the weights and decrease $t_{Pos_{\vec{\sigma}}[v]+1}$. When $Pos_{\vec{\sigma}}[f(\vec{x})] = Pos_{\vec{\sigma}}[v]$ 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 $\vec{\sigma}$. Notice that $\vec{\sigma}$ is known and given as input to the algorithm.

Our algorithm can learn any $\vec{\sigma}$-separable function as long as $\vec{\sigma}$ 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. $\vec{\sigma}$-separable, for some $(s+1,k)$-permutation $\vec{\sigma}$). In fact, the permutably homogeneous algorithm generalizes the homogeneous algorithm in that $\vec{\sigma}$ is any permutation on K (such permutation does not need to be the identity permutation nor a k-permutation).

When $\vec{\sigma}$ 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.

5.2.2 Permutably homogeneous (n,k,s)-perceptrons convergence properties

Given a $(s+1,k)$-permutation $\vec{\sigma}$, experiments show that the permutably homogeneous (n,k,s)-perceptron learning algorithm always converges for $\vec{\sigma}$-separable functions. That is, given the appropriate output vector any permutably homogeneous k-valued function will be learned. In this section we give a formal proof of convergence of the algorithm given in Figure 5.1.

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 [Obradović 94] that the homogeneous $(k,k-1)$-perceptron learning algorithm always terminates in learning homogeneous k-valued 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 π. 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_π 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 5.2.1 *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_π.*

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 $t_i \leq \vec{w}\vec{x} \leq t_{i+1}$ with $t_0 = -\infty$ and $t_k = +\infty$). So $\vec{r}_\pi$ is a $(k-1)$-representation for f_π. $\Leftarrow$) Similar proof using the fact that, for $z \in K$, we have $\sigma(z) = \pi^{-1}(\pi(z)) = z$ where the permutation π^{-1} is the inverse of π (π^{-1} is guaranteed to exist and is unique since the set of all permutations on K forms a group). •

Lemma 5.2.1 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_π 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 3.6 the three hyperplanes remain invariant even after transformation of f into $f_{(1,3,2,0)}$ and vice versa.

Theorem 5.2.1 *Given the output vector $\vec{\pi}$, the permutably homogeneous $(n, k, k-1)$-perceptron learning algorithm for learning a function $f \in P_k^n$ terminates if and only if f is $\vec{\pi}$-separable.*

Proof $\Rightarrow$) If the permutably homogeneous $(k, k-1)$-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 5.2.1 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 5.2.1. The algorithm is guaranteed to terminate by the homogeneous $(n, k, k-1)$-perceptron convergence theorem of [Obradović 94]. •

If in Figure 5.1 we replace $\vec{o}$ by $\vec{\pi}$, then we clearly have that $Pos_{\vec{\sigma}} = \pi^{-1}$ 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 $s \leq k-1$. In the sequel we let $\vec{\nu} = (\nu(0), \ldots, \nu(s))$ be a $(s+1, k)$-permutation and $\vec{\nu}^c = (\nu^c(0) = \nu(0), \ldots, \nu^c(s) = \nu(s), \nu^c(s+1), \ldots, \nu^c(k-1))$ be the k-permutation (full permutation) obtained by adding $k-s$ coordinates to $\vec{\nu}$.

Lemma 5.2.2 *f is $\vec{\nu}$-separable if and only if it is $\vec{\nu}^c$-separable.*

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

Theorem 5.2.2 **(Permutably homogeneous perceptron convergence theorem)** *Given the output vector $\vec{\nu}$, 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 $\vec{\nu}$-separable.*

Proof $\Rightarrow$) Same as in Theorem 5.2.1. $\Leftarrow$) The algorithm learns f using the output vector $\vec{\nu}^c$ instead of $\vec{\nu}$. Since from Lemma 5.2.2 f is $\vec{\nu}^c$-separable, then by Theorem 5.2.1 the algorithm is guaranteed to terminate. •

5.2.3 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 $L = \{4 <_L 1 <_L 3 <_L 0 <_L 2\}$ are two linear extensions of the poset shown in Figure 5.2. 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 $x <_P z <_P y$.

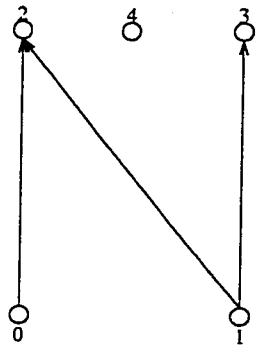


Figure 5.2: Example of partially ordered set.

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 5.3. 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 $1 \leq d \leq n$. 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 5.4 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 $(K_f, <_{K_f}^d)$

Procedure ExtendedLearning(f, n, k, s);
  $K_f \subseteq K$:= 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
      $\vec{e} := (e_0, \ldots, e_{|K_f|-1})$ the unique linear extension of $(K_f, <^d_{K_f})$;
      $\vec{p} := (p_0, \ldots, p_{s-|K_f|})$ a $(s - |K_f| + 1, |K - K_f|)$-permutation of $K - K_f$;
      $\vec{o} := (e_0, \ldots, e_{|K_f|-1}, p_0, \ldots, p_{s-|K_f|})$;
      MultiPerceptron$(f, n, k, s, \vec{o})$;
  Until *Contradiction* = True or *Unique* = True or $d = n$;
  If *Contradiction* = False and *Unique* = False then
    CombinePartialOrders$((K_f, <^1_{K_f}), \ldots, (K_f, <^n_{K_f}))$;

Figure 5.3: Extended (n, k, s)-perceptron learning algorithm.

for some d, or there is some d such that a poset $(K_f, <^d_{K_f})$ cannot be obtained, or $d = n$. When n non-chain posets $(K_f, <^1_{K_f}), \ldots, (K_f, <^n_{K_f})$ are constructed then, using the *partial orders combination algorithm* of Figure 5.5, 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 $1 \leq d \leq n$, we start with an antichain $(K_f, <^d_{K_f})$. Then, for every $\vec{x} = (x_1, \ldots, x_n)$, we construct poset $(K_f, <^d_{K_f})$ by adding new comparable pairs $f_1 = f(x_1, \ldots, x_d, \ldots, \ldots, x_n) <^d_{K_f} f(x_1, \ldots, x_d + 1, \ldots, x_n) = f_2$ whenever $f_1 \not<^d_{K_f} f_2$, and also, we add new comparable pairs $y <^d_{K_f} f_2$ whenever $y <^d_{K_f} f_1$ and comparable pairs $f_1 <^d_{K_f} y$ 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

```
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 5.4: Partial order construction algorithm.

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 origin is $\vec{y}$.

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 5.5: 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 5.6: Extension algorithm

Figure 5.8 shows examples of constructed posets. For illustration purpose, we have also shown the graphs obtained from function $g(x, y)$ (Figure 5.8b) 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.

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 5.7: Cutting algorithm

For given permutably homogeneous and s-separable function $f \in P^n_k$, 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 5.8c 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 $1 \leq d \leq n$, 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 $c = [c_1 \ldots c_d] \in \{0, 1\}^d$, where $1 \leq d \leq n$. 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 be both 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 $\leq n$ in lexicographic order. For example, for $n = 4$, the lexicographic generation of binary strings of lengths ≤ 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 $r \leq d$ such that $c_r = 0$. We remove posets $(K_f, <^r_{K_f})$ and $(K_f, >^{r<i\leq d}_{K_f})$ 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 5.9 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 $(K_f, <_{K_f}) \ominus (K_f, <^d_{K_f}) = \bigcup_{[c_1 \ldots c_i \ldots c_{d-1}]} (K_f, <^i_{K_f})$ (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 5.8 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 5.8a), 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 5.8b) 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 5.8c), 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 5.9 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 5.2.3 *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 $1 \leq d \leq n$.*

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. •

Some non-thick s-separable functions may have all combination posets $(K_f, <_{K_f})$ to be non-chains (thus they have many linear extensions) and so the last *repeat* loop of the algorithm in Figure 5.5 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 $O(k^n|K_f|)$.

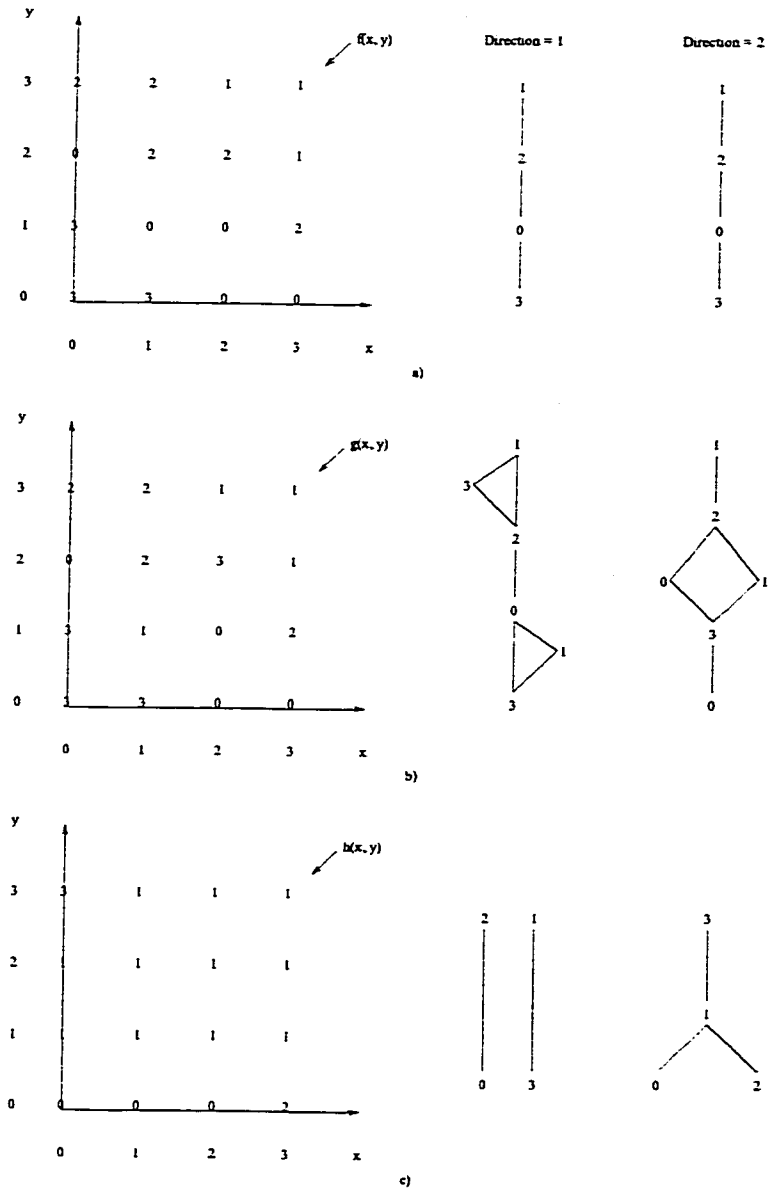


Figure 5.8: Constructed partial orders for some $f \in P_4^2$.

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 ex-

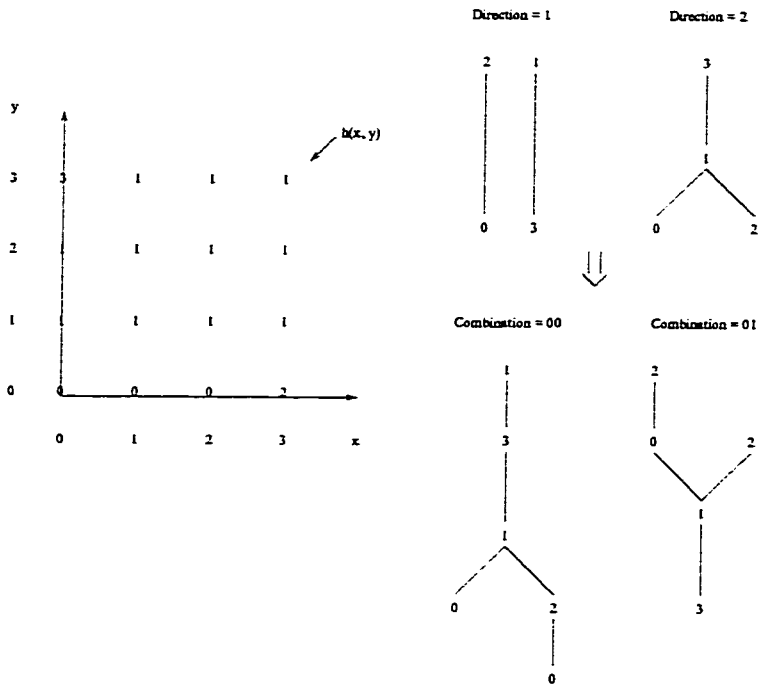


Figure 5.9: Examples of combinations posets for Figure 5.8c.

tension 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 $2^n \leq k^n$ and $|K_f|^2 \leq |K_f|!$ 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 $f \in P_k^n$ 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.

5.3 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 η 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 5.9. 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 $(n \geq 3)$-place functions.

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

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

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.

5.4 Conclusion

In this chapter we have discussed an extended learning algorithm for learning the class of permutably homogeneous multiple-valued 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.

Another interesting research problem is to develop learning algorithm for multilayer neural networks whose processing units are (n, k, s)-perceptrons. Such networks would have the ability to learn any k-valued function.

Chapter 6

Synthesis of Multiple-Valued Functions Based on the Longest Strip or the Maximum Separable Subset

We consider the problem of synthesizing multiple-valued functions by neural networks. A genetic algorithm which finds either the longest strip or the maximum separable subset for a subset in $V \subset K^n$, by using different fitness functions, is described. A strip contains points located between two parallel hyperplanes. Repeated application of the genetic algorithm 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 compared with other methods such as tabu search.

6.1 Introduction

In this chapter, we propose to synthesize k-valued logic functions by *minimal* multilayer feed-forward neural networks. Multilayer feed-forward neural networks [Rumelhart 86] are in principle able to learn any arbitrary mapping, provided that enough hidden units are present [Minsky 69]. For these networks, learning algorithms such as back-propagation [Rumelhart 86] have been found to be computationally prohibitive [Denker 87]. Also, the topol-

ogy of the network must be fixed before learning.

This brings the problem of network initialization, that is to construct first a depth-optimal (number of layers) or size-optimal (number of processing units) or depth&size-optimal network. 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 [Judd 87]. Also, it has been shown in [Blum 88] that the problem of finding the absolute minimal architecture for a given task is NP-hard. A third problem, known to be NP-complete [Siu 95], which also concerns us here is the following. Given two sets $S_1, S_2 \subseteq R^n$ such that $|S_1 \cup S_2| = c$, determines 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* [Siu 95] which is to find a linearly separable subset of maximum cardinality.

When applied to multiple-valued case, these three problem are also NP-complete since they are known to be NP-complete in the synthesis of two-valued functions and that two-valued functions are special cases of multiple-valued functions.

6.2 Existing solutions

Many heuristics such as *cascade correlation algorithm* [Fahlman 91], *upstart algorithm* [Frean 90], *entropy nets* [Sethi 90], *neural trees* [Sirat 90], *tiling algorithm* [Mezard 89], *regular partitioning* [Ruján 89], and other partitioning algorithms [Gallant 86, Golea 90, Marchand 93a, Marchand 90, Nadal 89], are proposed as approaches of building networks which are (near) optimal for a given task. These heuristics are known as *constructive* or *growth algorithms* since they all construct a network starting from zero or a small number of hidden units.

In this paper we introduce two methods of partitioning a set $V \subseteq K^n$ using a genetic algorithm to grow a multiple-valued logic neural network for learning a function. Our main problem is to find a partition of the set V which uses the minimum possible number of hyperplanes not necessarily parallel. Our partitioning approach is an extension of the method proposed by Marchand et al. [Marchand 90] to the multiple-valued logic case. Their partitioning algorithm construct a neural network of linear threshold units with one hidden layer. In the next paragraphs we describe the essence of a

partitioning algorithm.

In [Marchand 90], 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 halfspace constructed by their algorithm be $H_1, \ldots, H_r$ and define u_i to be 0 (resp. 1) if f is 0 (resp. 1) on the region cut off by H_i for $1 \leq i \leq r$. Then adding an output unit with threshold 0 and weight $u_i 2^{r-i}$ for the edge leaving the hidden unit corresponding to H_i, they get a neural network computing f. They assume that linear threshold units produces an output 0 or 1. In a restricted version of their approach, called *regular partitioning* [Ruján 89], 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 the units are connected (in term of network weights and topology). Experiments indicate that partitioning algorithms are successful in the sense that they construct small neural networks efficiently [Keibek 92, Marchand 93a, Marchand 90, Rivest 87].

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* halfspace. 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 *neural decision list* or *linear decision list* [Marchand 93a, Rivest 87, Turán 96].

More formally, [Turán 96] 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 0 or 1 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 $m+1$.

Partitioning techniques such as cascade correlation [Fahlman 91], upstart algorithm [Frean 90], entropy nets [Sethi 90] and tiling [Mezard 89] apply only for Boolean output functions, that is for functions $f : R^n \mapsto \{0,1\}$. The others partitioning methods such as neural trees [Sirat 90], regular partitioning [Ruján 89] and neural decision lists [Gallant 86, Golea 90, Marchand 93a, Marchand 90, Nadal 89] apply for Boolean output functions as well as for multiclass functions $f : R^n \mapsto K$ (k-valued functions are multiclass functions). No results have been reported, however, in literature for multiclass functions when a growth algorithm is used.

Techniques are known in literature for learning multiclass functions. The most powerful one is the *error-correcting output codes* (or $ECOC$) approach [Dietterich 95, Schapire 97] which is a robust method of solving multiclass learning problems by reducing to a sequence of two-class problems. It is shown in [Dietterich 95] that ECOC learning provides a general-purpose method for improving the performance of inductive learning programs on multiclass problems. [Obradović 94] describes a learning algorithm for k-valued logic functions on either a single $(n, k, k-1)$-perceptron or a depth-two network composed of k $(n, 2, 1)$-perceptrons in the hidden layer and one $(k, k, k-1)$-perceptron as output unit. Other simpler methods use back-propagation (or any appropriate learning algorithm such as adaline or radial basis function, etc) networks with multiple output units assigned to distinct classes. All these approaches use fixed-architecture networks for learning k-valued functions and thus a given network may not be optimal for an arbitray problem.

6.3 Longest strip and maximum separable subset based growth algorithms

In our particular implementation of partitioning algorithm, genetic algorithm (GA) and tabu search (TB) are, respectively, used to obtain subsequent halfspaces delimited by either one or two hyperplanes (depending on a predefined subset G). To each halfspace we assign a hidden unit that correctly classifies all elements of it. Our growth algorithm differs from the techniques mentioned above in the fact that the (near) minimal partition is found first before the neural network is constructed (using the partition), whereas in the other methods, the network is being constructed while the

set of examples is being partitioned.

In this section we describe two methods of synthesizing multiple-valued logic functions by removal of points in a predefined subset $G \subseteq V$. In the first method G is defined as the longest strip of training examples whereas in the second method it represents the maximum separable subset of training examples. For $k = 2$, the second method is an alternative to the solution suggested by Marchand [Marchand 93a].

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. In both methods, we use a genetic algorithm (resp. tabu search) to find the subset G of currently given training examples. Examples of longest strip and maximum separable subset are shown, respectively, in Figure 6.1 and 6.2. Figure 6.3 shows our main growth algorithm for either synthesis methods.

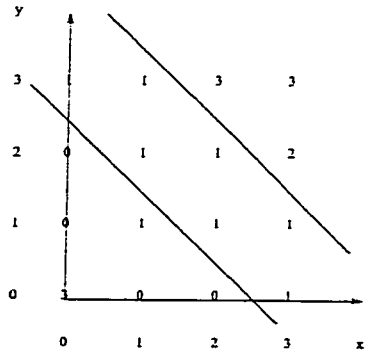


Figure 6.1: Example of longest strip for $k = 4$ and $n = 2$.

The details of the G-based synthesis algorithm using GA are described in the sections below.

6.3.1 Problem representation

Fundamental to the genetic algorithm structure is the encoding mechanism for representing the problem's variables. For the problem of determining G, the search space is the space of weight vectors $\vec{w}$ (as in [Whitley 92]) and the representation is more complex. It seems natural to represent the possible solutions as vectors $\vec{w} \in R^n$ and design specialized genetic operators which are suitable for the G problem. Each weight vector will uniquely determine

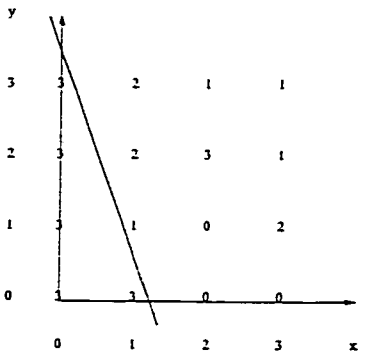


Figure 6.2: Example of maximum separable subset for $k = 4$ and $n = 2$.

Procedure G-BasedSynthesis(n, k, f);
  $r := 1$;
  $S := V$;
  Repeat
    Apply GA (or TB) to find subset G of S;
    Create a new hidden unit U_r with respect to G;
    $S := S - G$;
    $r := r + 1$;
  Until $S = \emptyset$;
  Construct a network with the r hidden units on the first layer;

Figure 6.3: G-based synthesis algorithm.

a partition of $V \subseteq K^n$ into $s + 1$ classes with s parallel hyperplanes (for some s) and the best weight vector is the one which generates subset G. To determine how good is a solution the GA needs a fitness function to evaluate each chromosome $\vec{w}$.

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

A note on the initial population. We initialize the population with ran-

dom 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 $1 \leq i \leq n$) for each vector $\vec{w}$, where α_i is a random number in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Initial population should consists of random hyperplanes $\vec{w}$.

6.3.2 Fitness function

The objective function, the function to be optimized, 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 array 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, $P_i = (\vec{w}\vec{x}_i, f(\vec{x}_i))$, $1 \leq i \leq |S|$, where $\vec{w}\vec{x}_i \leq \cdots \leq \vec{w}\vec{x}_{|S|}$. A strip in S is a sequence $P_i P_{i+1} \cdots P_{i+j}$ 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 $1 \leq i \leq |S|$ and $0 \leq j \leq |S| - i$. The length of the strip is $j - i + 1$. For example, in Figure 6.1 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 us the longest strip $P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13}$.

Given a set of points $S \subseteq K^n$ and a function f over S, let $S_1 = P_1 \cdots P_{j_1}$ and $S_2 = P_{j_2} \cdots P_{|S|}$ $(1 \leq j_1 \leq j_2 \leq |S|)$ be respectively the first and last strips generated by $\vec{w}$. We denote by $L(S, \vec{w})$ the length of the longest such strip generated by $\vec{w}$ and denote by $M(S, \vec{w})$ the length of the maximum between strips S_1 and S_2, 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 subset G.

- G = longest strip

$$Fitness_L(\vec{w}) = \frac{L(S, \vec{w})}{|S|} \tag{6.1}$$

- G = maximum separable subset

$$Fitness_M(\vec{w}) = \frac{M(S,\vec{w})}{|S|} \tag{6.2}$$

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 $Fitness_{(L \text{ or } M)}(\vec{w})$ 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 $O(pnk^n \log k)$ 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 $\frac{p}{2}$ new chromosomes are evaluated for their fitness and hence, our GA has a time complexity of $O(gpn|S|\log|S|) \approx O(gpnk^n \log k)$.

6.3.3 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, with equal probability, two of the following three crossovers operations.

$$\vec{c}_1 = \vec{p}_1 + \vec{p}_2 \tag{6.3}$$

$$\vec{c}_1 = \vec{p}_1 - \vec{p}_2 \tag{6.4}$$

$$c_{2_i} = \begin{cases} p_{1_i} & \text{if } random() \leq 0.5 \\ p_{2_i} & \text{otherwise} \end{cases} \tag{6.5}$$

Crossover in (6.3) 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 (6.4) is the substraction of two parents and the child is the vector orthogonal to the sum of its parents. Crossover in (6.5) is a uniform crossover of two parents, that is, at coordinate i each parent have 50% chances to be selected as c_{i_j} $(1 \leq j \leq n)$. 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 children are also normalized to unit vectors after any crossover or mutation operation.

6.3.4 Mutation

After crossover, chromosomes are subjected to random mutations. We propose three methods of *coordinate-wise* mutations as described. They correspond to 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:

$$c_i = random[-1, 1] \tag{6.6}$$

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:

$$c_i = \pm\sqrt{1 - p_i^2} \tag{6.7}$$

Neighborhood 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:

$$c_i = p_i \pm \frac{m}{k^n} \tag{6.8}$$

where $m \leq k$ 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 $\frac{1}{3}$.

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 $p_{muta} \leq 0.1$.

6.3.5 Constructing the neural network

G = longest strip

At every iteration r of (6.3) the genetic algorithm finds a chromosome $\vec{w}_r$ which produces the longest strip $P_iP_{i+1}\cdots P_{i+j}$ ($1 \leq i \leq |S|$ and $0 \leq j \leq |S|-i$) with strip value $v = f(\vec{x}_i)$. Let $v_r = v+1$. 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 $\vec{o}_r = (0, v_r, 0)$. In other words, the perceptron has a transfer function of the form $g_{k+1,2}^{(\vec{w}_r\vec{x}_i, \vec{w}_r\vec{x}_{i+j+1}),(0,v_r,0)} : S \mapsto \{0, v_r\}$ (that is a $(k+1)$-valued two-threshold function). Let T be the set of all points contained in the longest strip, then the $(n, k+1, 2)$-perceptron will output the value v_r for all points $\vec{x} \in T$ and will output the value 0 for all points $\vec{x} \in S - T$.

G = maximum separable subset

At every iteration r of (6.3) the genetic algorithm finds a chromosome $\vec{w}_r$ which produces a maximum separable subset we denote by $M_r(\vec{w}_r) = \max(S_1, S_2)$. If $M_r(\vec{w}_r) = S_1$ (resp. S_2), let $v_r = v_1 + 1$ (resp. $v_2 + 1$) where v_1 (resp. v_2) is the function value of all point in S_1 (resp. S_2). Then we create a $(n, k+1, 1)$-perceptron (hidden unit U_r) whose weight vector is $\vec{w}_r$, threshold vector is $\vec{t}_r = (\vec{w}_r\vec{x}_{j_1+1})$ if $M_r(\vec{w}_r) = S_1$ or $\vec{t}_r = (\vec{w}_r\vec{x}_{j_2})$ if $M_r(\vec{w}_r) = S_2$, and output vector is $\vec{o}_r = (v_r, 0)$ or $\vec{o}_r = (0, v_r)$ depending on $M_r(\vec{w}_r)$. In other words, the perceptron has a transfer function of the form $g_{k+1,1}^{(\vec{w}_r\vec{x}_{j_1+1}),(v_r,0)} : S \mapsto \{0, v_r\}$ or $g_{k+1,1}^{(\vec{w}_r\vec{x}_{j_2}),(0,v_r)} : S \mapsto \{0, v_r\}$ (that is a $(k+1)$-valued one-threshold function). The $(n, k+1, 1)$-perceptron will output the value v_r for all points $\vec{x} \in M_r(\vec{w}_r)$ and will output the value 0 for all points $\vec{x} \in S - M_r(\vec{w}_r)$.

Once we have defined all units $U_1, \ldots, U_r$, (r = number of runs of the GA —in the r-th call only the fitness function is executed once) then our next

step is to construct a feed-forward multilayer neural network. We propose two network topologies.

6.3.6 Depth four, $r + k + 2$ units, $3r + k - 1$ thresholds architecture

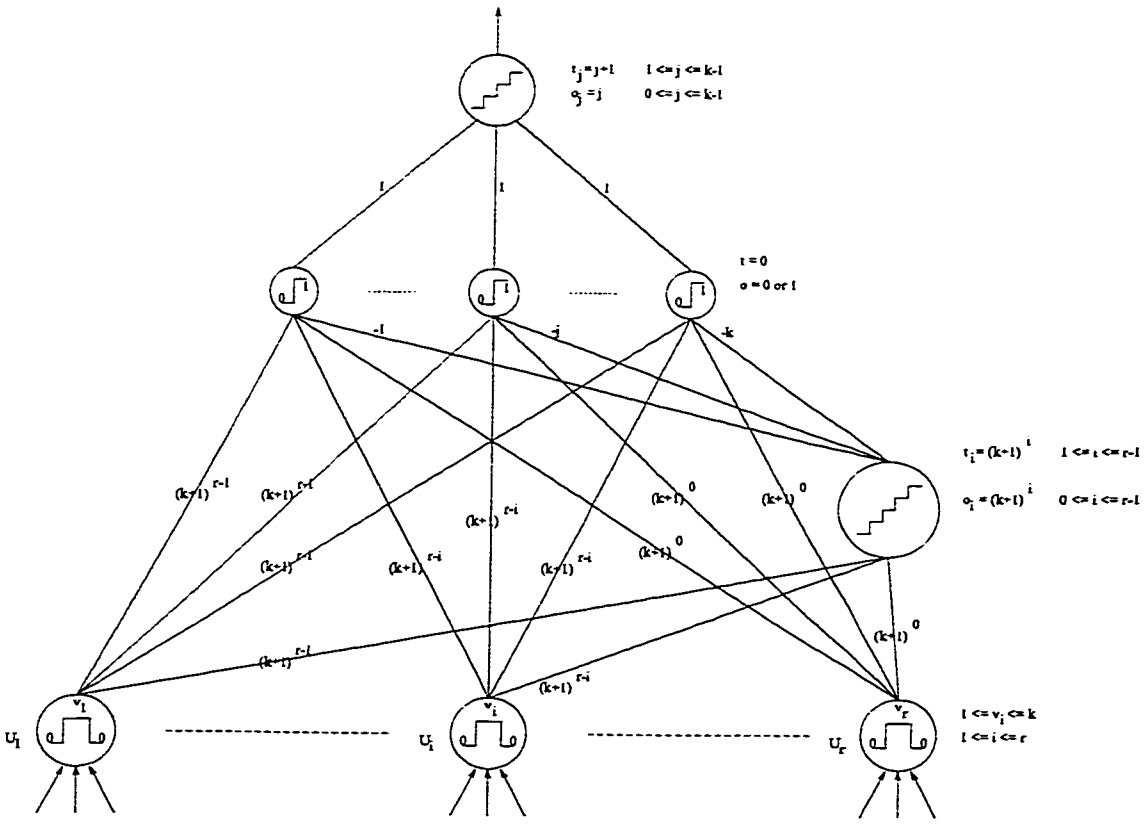


Figure 6.4: Network with four depths, $r + k + 2$ units, $3r + k - 1$ thresholds.

The network in Figure 6.4 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 G, and there are r such units (Figure 6.4 shows the case for G = longest strip).

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

Hidden layer 3 contains k units. Each unit of layer 1 and 2 are 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 0.

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 1.

The total number of thresholds in the network is $2r+r-1+k = 3r+k-1$.

6.3.7 Depth two, $r+1$ units, $2r+kr$ thresholds architecture

The network in Figure 6.5 is equivalent to Marchand's construction [Marchand 90] but generalized to multiple-valued logic functions. It has one hidden layer and $r+1$ neurons. Hidden layer 1 is same as in Figure 6.4 (Figure 6.5 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 $\vec{w}$.

The total number of thresholds in the network is $2r+kr$.

6.3.8 Proof of correctness

In this section we prove that both networks, for both definitions of G, effectively computes any given but arbitrary logic function $f \in P_k^n$.

Let $\vec{x}$ be an input vector applied to the network (we refer to both networks and both definitions of G). Let $\vec{u} = (u_1, \ldots, u_r)$ be the set of output

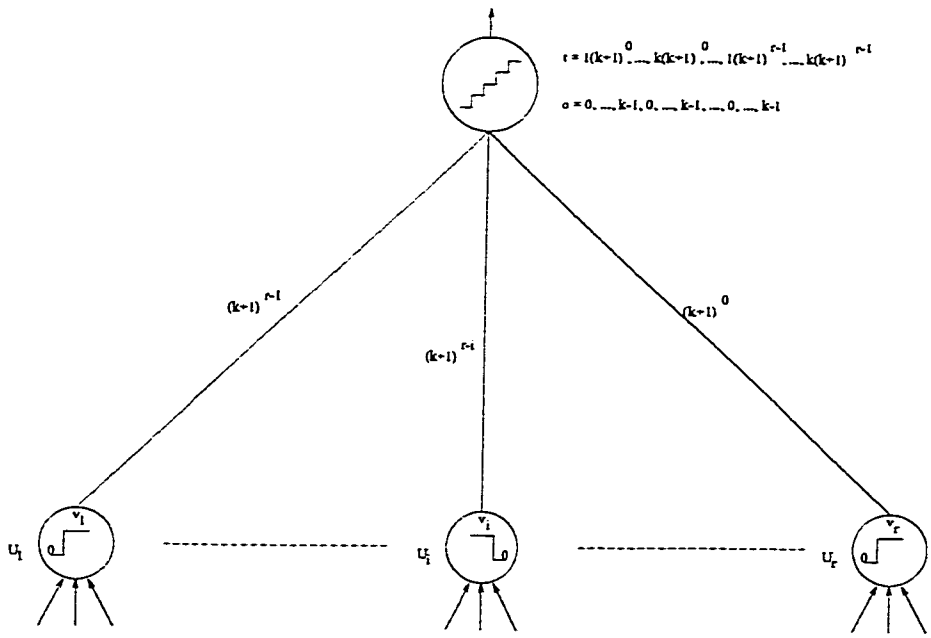


Figure 6.5: Network with two depths, $r+1$ units, $2r+kr$ thresholds.

values of the units of layer 1 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 2. Let $\vec{q} = (q_1, \ldots, q_k)$ be the outputs of layer 3, that is unit Q_j has value $q_j \leq 1$ on input $\vec{x}$ $(1 \leq j \leq k)$. 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 Figures 6.4 and 6.5). Recall that each U_i corresponds to a subset G_i (the longest strip —or the maximum separable subset) found by the genetic algorithm at its i-th run and removed by our main algorithm. The collection of the G_i's is a partition of V, that is $\bigcup_{i=1}^{r} G_i = V$ 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 1 (starting from the left) which outputs a non-zero valuer 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 0 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 3 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 1 and layer 3 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 3 will output the value 1.

By definition, the output unit Z which has only unit weights, will perform the sum $c = \sum_{j=1}^{k} 1b_j = \sum_{j=1}^{u_i} 1 = u_i$. 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 Figure 6.4. The proof for Figure 6.5 is straightforward.

6.4 Hill-climbing method for the longest strip and the maximum separable subset problems

We also designed a *hill-climbing* technique called *tabu search* [Glover 93] for the longest strip and maximum separable subset problems. The G-based synthesis algorithm shown in Figure 6.3 is used. However instead of using GA to find the best weight vector, we use TB to move to the best neighbor of a current weight vector. Our algorithm is shown in Figure 6.7.

Our idea is as follows. We initially start with a unique random unit weight $\vec{w}_0 = (w_{0,1}, \ldots, w_{0,n})$. Then the next weight vector $\vec{w}_1 = (w_{1,1}, \ldots, w_{1,n})$ is chosen to be a neighbor of $\vec{w}_0$ which has the best fitness value higher than the fitness of $\vec{w}_0$ and which yields the least change in the ordering of the function values according to $\vec{w}_0$ (see section 6.3.2). Then we set $\vec{w}_2$ to be the best neighbor of $\vec{w}_1$, ..., and so on until there is no more best neighbor or a certain number of iterations is reached.

Recall that to find the subset G for a current $\vec{w}$ and a set of points S with $|S| = v$ we sort the v points with respect to $\vec{w}$ (first key) and $f(\vec{x})$ (second key) and find the longest strip or the maximum separable subset from the sorted list $(\vec{w}\vec{x}_1, f(\vec{x}_1)), \ldots, (\vec{w}\vec{x}_v, f(\vec{x}_v))$. The neighbors of $\vec{w}$ are precisely those weight vectors that yield the least change in the ordering of the $f(\vec{x})$'s.

For instance, if $v = 4$ and $\vec{w}_0$ gives the order $(f(\vec{x}_1), f(\vec{x}_2), f(\vec{x}_3), f(\vec{x}_4))$,

then a close neighbor of $\vec{w}_0$ may produce the order $(f(\vec{x}_2), f(\vec{x}_1), f(\vec{x}_3), f(\vec{x}_4))$ and a far neighbor of $\vec{w}_0$ may produce the order $(f(\vec{x}_4), f(\vec{x}_3), f(\vec{x}_2), f(\vec{x}_1))$.

Consider the j-th coordinate, $w_{0,j}$, of $\vec{w}_0$, $1 \leq j \leq n$. Let $\vec{w}_0\vec{x}$ and $\vec{w}_0\vec{y}$ be two consecutive elements in the order produced by $\vec{w}_0$ and let $d = \vec{w}_0\vec{y} - \vec{w}_0\vec{x} = \vec{w}_0(\vec{y} - \vec{x}) = w_{0,j}(y_j - x_j) + \sum_{i \neq j} w_{0,i}(y_i - x_i)$. Likewise, for an arbitrary vector $\vec{w}'_0$, we will have $d' = w'_{0,j}(y_j - x_j) + \sum_{i \neq j} w'_{0,i}(y_i - x_i)$.

Vector $\vec{w}'_0$ is a neighbor of $\vec{w}_0$ with respect to coordinate j if $w'_{0,i} = w_{0,i}$ for $i \neq j$ and $|w'_{0,j} - w_{0,j}|$ is minimal such that the sorted order is changed. Then only $w_{0,j}$ is affected and hence $\sum_{i \neq j} w_{0,i}(y_i - x_i) = \sum_{i \neq j} w'_{0,i}(y_i - x_i)$. Since $\sum_{i \neq j} w_{0,i}(y_i - x_i) = d - w_{0,j}(y_j - x_j)$ therefore we have $d' = w'_{0,j}(y_j - x_j) + d - w_{0,j}(y_j - x_j) = d + \Delta w_{0,j}(y_j - x_j)$, where $\Delta w_{0,j} = (w'_{0,j} - w_{0,j})$. Taking $d' = 0$ and solving for $\Delta w_{0,j}$ we obtain

$$\Delta w_{0,j} = -\frac{d}{y_j - x_j} \tag{6.9}$$

Denote the set of neighbors of $\vec{w}_0$ with respect to j by $N_{0,j}$ and the set of all neighbors of $\vec{w}_0$ by $N_0 = \bigcup_{j=1}^{n} N_{0,j}$. To obtain the set $N_{0,j}$ of $\vec{w}_0$ we compute the differences $d_i = \vec{w}_0(\vec{x}_{i+1} - \vec{x}_i)$ for $1 \leq i \leq v - 1$, where $\vec{w}_0\vec{x}_i$ and $\vec{w}_0\vec{x}_{i+1}$ are two consecutive elements in the sorted list produced by $\vec{w}_0$. Let $\vec{w}^i_0 \in N_{0,j}$ be a neighbor of $\vec{w}_0$, then we have $w^i_{0,j} = w_{0,j} + \Delta w_{0,j_i}$ and $w^i_{0,c} = w_{0,c}$ for $c \neq j$, where $\Delta w_{0,j_i} = -\frac{d_i}{x_{i+1,j} - x_{i,j}}$. In other words, the Euclidean distance, $D(\vec{w}_0, \vec{w}^i_0)$ (or D_{0,j_i} for short), between $\vec{w}_0$ and $\vec{w}^i_0$ is $|\Delta w_{0,j_i}|$. Thus the set N_0 can be defined as

$$N_0 = \{\vec{w}^i_0 \mid D_{0,j_i} = |\Delta w_{0,j_i}|, 1 \leq i \leq v - 1, 1 \leq j \leq n\} \tag{6.10}$$

A nearest neighbor of $\vec{w}_0$ is a vector $\vec{\omega} \in N_0$ such that $D(\vec{w}_0, \vec{\omega}) = Min\{D_{0,j_i}, 1 \leq i \leq v - 1, 1 \leq j \leq n\}$.

Now that we know how to obtain a neighbor $\vec{w}$ of $\vec{w}_0$, the main remaining task is to decide how to select $\vec{w} \in N_0$ for $\vec{w}_1$. One solution is to always select the nearest neighbor of $\vec{w}_0$ in N_0 for $\vec{w}_1$ and whose fitness is the best so far. Another solution is to set $\vec{w}_1$ to be the element of N_0 which has the highest fitness value greater than that of $\vec{w}_0$. If no such element exists, then we have a local maximum and we must decide how to set $\vec{w}_1$ in order to escape the local maximum. We will describe later in this paragraph some solutions to this problem.

However, doing an exhaustive search in N_0, all the time, may not be efficient even though $|N_0|$ is linear in the number of points v. This because

N_0 can be very large in the beginning of our constructive algorithm, depending on k, n and $S \subseteq K^n$ (where $|S| = v$). In fact, each coordinate of $\vec{w}_0$ produces $v-1$ neighbors at most (a coordinate will not produce a neighbor if $x_j = y_j$ or $d = 0$, in (6.9)), and hence $|N_0| \leq n(v-1) \leq n(k^n - 1)$.

Our alternative approach is the following. We randomly select an integer $j \in \{1, n\}$ and compute the set $N_{0,j}$ of neighbors of $\vec{w}_0$ with respect to j. Then we do an exhaustive search in $N_{0,j}$ to set $\vec{w}_1$ to the fittest element of $N_{0,j}$ fitter than $\vec{w}_0$ and restart the whole process again in order to compute $\vec{w}_2$, and so on. This process terminates if either there can be no more improvement on the current best solution or a maximum number of generation is reached.

At generation i, let $\vec{b}_{N_{i,j}}$ be the best weight vector in $N_{i,j}$ and $\vec{b}$ be the best solution obtained so far. If $f(\vec{b}_{N_{i,j}}) \leq f(\vec{b})$ then $\vec{b}_{N_{i,j}}$ and $\vec{b}$ are both local maxima. One way to set $\vec{w}_{i+1}$ before we move to the next generation is the following. We first attempt to search in c sets $N_{i,\iota}$ (c is a random integer in $\{1, n-1\}$ and $\iota \neq j$). If the search in one set $N_{i,\iota}$ is successful then we move to generation $i+1$, otherwise we search the remaining $N_{i,\iota}$'s until success. If all $N_{i,\iota}$'s have been searched with no success at all, then we set $\vec{w}_{i+1}$ to $\vec{b}$ or $\vec{w}_i$ or $\vec{b}_{N_{i,j}}$ (or the best of all $N_{i,\iota}$'s) and randomly alter values at some coordinates of $\vec{w}_{i+1}$ before we move to the next generation. The alteration of $w_{i+1,j}$, for $1 \leq j \leq n$, is done as follows: with a probability of 0.5, we set $w_{i+1,j} = w_{i+1,j} \pm \frac{a}{v}$ where a is a random real in $[1, k]$.

To avoid cycling and oscillation of weight vectors (that is a solution that has been tested already may appear many times) we implement a tabu search method as follows. At each iteration i, $\vec{w}_i$ is the current solution and $\vec{w}_{i+1}$ is the next solution to obtain. Suppose that $\vec{w}_{i+1}$ is computed according to above paragraphs, then before we move to the next iteration we must make sure that $\vec{w}_{i+1}$ *is not in* a *tabu list* containing weight vectors which are already obtained in past iterations. If $\vec{w}_{i+1}$ *is in* the tabu list then we must find another candidate for $\vec{w}_{i+1}$ which is not *tabu* and move to the next iteration. Therefore in this way we prevent generating a *same* solution again. We use a tabu list of kn already tested solutions, and every time we generate a new solution, we put it in the tabu list if it is not tabu. Although there are many ways to update a tabu list, we use the following: we select a random position in the list and replace the element at that position with the new solution.

We mean by the terms '*same*' or '*is in tabu list*' the fact that a candidate solution $\vec{w}$ to consider for next generation is *equivalent* to an element $\vec{\omega}$ in the tabu list (we also say that $\vec{\omega}$ is tabu, and so is any equivalent vector to

it). Two solutions are equivalent, or similar, if they both produce the same order on the $f(\vec{x})$'s, otherwise they are distinct. We use the dot product $\vec{w}\vec{\omega}$ as a measure of similarity between two vectors. We have that $\vec{w}\vec{\omega} = \|\vec{w}\|\|\vec{\omega}\| \cos\alpha = \cos\alpha$ (since both $\vec{w}$ and $\vec{\omega}$ are unit vectors), where $\alpha = \widehat{\vec{w},\vec{\omega}}$ is the angle between the two vectors. Thus the smaller is α the closer are the vectors. For the vectors to be equivalent, angle α must be strictly smaller than the least possible angle for which $\vec{w}$ and $\vec{\omega}$ are distinct. Denoting such angle β, we show in next paragraph how to obtain β.

Consider Figure 6.6 for instance. Angle β is in fact the angle between the hyperplane P_1 passing through the two farthest points of the n-cube K^n (A and B in the figure) and the hyperplane P_2 intersecting P_1 at one endpoint (A in the figure) and passing through a point which is at distance one from the other endpoint (B in the figure) and which lies in the boundary of K^n.

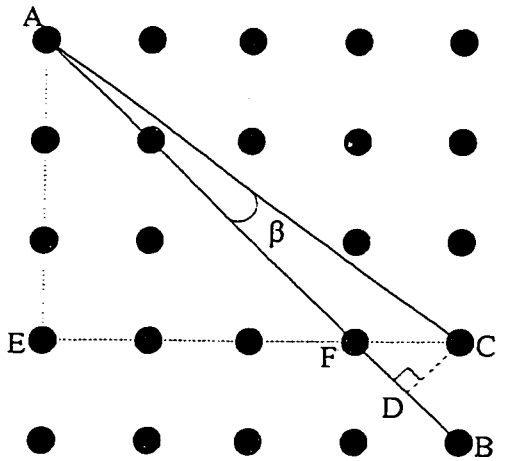


Figure 6.6: Smallest angle in K^n with $k = 5$ and $n = 2$.

Using elementary geometry we have $\cos\beta = \frac{AD}{AC} = \frac{AF+FD}{AC} = \frac{AF+\frac{FB}{2}}{AC}$, which gives

$$\cos\beta = \frac{(2k-3)\sqrt{2}}{2\sqrt{(k-2)^2+(k-1)^2}} \tag{6.11}$$

Therefore $\vec{w}$ and $\vec{\omega}$ will be equivalent if and only if $\vec{w}\vec{\omega} > \cos\beta$, otherwise they are distinct.

We have one last problem to solve: what to do when at generation i $\vec{w}_{i+1}$ is tabu? One solution is to search in a random number of $N_{0,\iota}$ (where ι is not yet used) until we obtain a candidate for $\vec{w}_{i+1}$ which is not tabu. If after

```
Procedure HillClimbing(S, k, n, f);
  cos β := (2k−3)√2 / 2√((k−2)²+(k−1)²); {used to check for tabuness}
  v := |S|;
  i := 0;
  w_i := random unit vector;
  Insert w_i in tabu list;
  b := w_i;
  Repeat
    Select random index j ∈ {1, n};
    w_{i+1} := b_{N_{i,j}};
    If w_{i+1} is not tabu and f(w_{i+1}) > f(b) then
      b := w_{i+1};
    Else
      t := random number in {1, n − 1}; {maximum number of trials}
      Repeat
        Select random index ι ∈ {1, n} not yet tried;
        w_{i+1} := b_{N_{i,ι}};
        If w_{i+1} is not tabu and f(w_{i+1}) ≥ f(b) then
          b := w_{i+1};
          success := true;
        Else
          w_{i+1} := max(w_{i+1}, b_{N_{i,ι}});
      Until success or t N_{i,ι}'s are tried;
      If no success then
        If w_{i+1} is tabu and f(w_{i+1}) ≥ f(b) then
          With probability of 50% chance to alter or not to alter do
            w_{i+1} := random alteration of w_{i+1};
        Else
          With equal probability or not do either
            1: For an ι not yet selected {last trial}
                 w_{i+1} := max(w_{i+1}, b_{N_{i,ι}});
            2: w_{i+1} := random alteration of b;
            3: w_{i+1} := random alteration of w_{i+1};
            4: w_{i+1} := random unit vector; {big jump}
    Insert w_{i+1} in tabu list;
    i := i + 1;
  Until Stopping criteria is true;
```

Figure 6.7: Hill-climbing to find G.

a certain number of trials $\vec{w}_{i+1}$ is still tabu, then with equal probability, we either decide to let $\vec{w}_{i+1}$ to be the next current solution (so we are backtracking), or we alter $\vec{w}_{i+1}$ and use it as current solution for generation $i+1$.

Our hill-climbing algorithm is shown in Figure 6.7. In the algorithm $\vec{b}$ is the best solution so far, $\vec{w}_i$ is the current solution at generation i and $\vec{b}_{N_{i,j}}$ is the best solution in $N_{i,j}$ with respect to index j.

6.5 Experiments and discussions

To analyze the performance of our methods, we tested each method on four classes of functions: random functions, random linear functions, random monotone functions and permutably homogeneous functions. We designed two sets of experiments. In the first set, the training set is the set K^n (for given k and n), whereas in the second set, we randomly select a fraction of 60% of points from K^n as training set and test the constructed networks on the remaining 40% testing set. The training error achieved in all experiments is always 0, however as we will see later, testing error varies from function class to function class. For each given class of function and method of network construction, we ran our algorithms ten times. All ten runs of the algorithms use the same functions (generated randomly) but with different random seeds.

Throughout the experiments, we used the same GA parameters, namely: 1000 generations, 0.75 for crossover rate and 0.005 for mutation rate. We also used an elitist strategy, 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 of 0.005. On the other hand we need this high mutation rate in order to search effectively in the infinite space of weight vectors.

Table 6.1 shows the results of tens runs of each method on functions from each test class. The functions were generated randomly and they are the same in each method. In each set of ten runs the same function is used but we reinitialized the random number generator with different random seed. In the table, TB stands for 'tabu search' whereas GA stands for 'genetic algorithm'. Tabu search is faster than genetic algorithm, therefore we used 5000 as the number of generations in the tabu search method, in order to have approximately the same running time for 1000 generations in the genetic algorithm. The table is divided into two parts. The first part consists of results obtained when using K^n as training set. In the second

		Using 100% of K^n		Using 60% of K^n		
		#Nodes	Time	#Nodes	Time	Accuracy
Rand.	GA Longest	28.60 ± 00.70	2915	19.80 ± 01.48	2399	23%
	TB Longest	30.00 ± 01.25	2362	20.00 ± 01.15	1557	25%
	GA Maximum	66.00 ± 02.71	6138	34.70 ± 04.00	4428	24%
	TB Maximum	67.60 ± 03.51	5351	33.80 ± 06.32	2692	26%
Line.	GA Longest	10.00 ± 00.00	858	14.70 ± 01.49	745	40%
	TB Longest	11.30 ± 01.34	789	14.70 ± 01.49	694	38%
	GA Maximum	20.80 ± 04.94	2271	41.60 ± 04.43	2972	22%
	TB Maximum	23.10 ± 06.40	1590	40.80 ± 04.57	2689	21%
Mono.	GA Longest	12.40 ± 01.43	1201	09.20 ± 01.03	635	59%
	TB Longest	12.30 ± 00.95	1142	09.20 ± 01.03	515	56%
	GA Maximum	14.40 ± 01.96	1261	10.60 ± 01.35	687	55%
	TB Maximum	14.60 ± 01.89	1070	10.60 ± 01.35	537	54%
Perm.	GA Longest	03.00 ± 00.00	321	06.20 ± 01.03	329	76%
	TB Longest	04.60 ± 00.97	308	06.20 ± 01.03	249	73%
	GA Maximum	03.00 ± 00.00	424	07.60 ± 01.34	348	81%
	TB Maximum	06.30 ± 01.34	330	07.00 ± 01.89	289	73%

Table 6.1: Results of 10 runs for $k = 4$ and $n = 4$.

part, we randomly select $60\%k^n$ points for training and the other $40\%k^n$ points for testing. In each part we display the average number of nodes —in the first layer of the constructed networks— with its standard deviation, and the average running time, over ten runs. The third column of the second part gives the average accuracy obtained over ten runs when testing the networks on the testing set.

Random functions are more difficult to learn because of the smaller separation amongs their inputs. Therefore it is not a surprise that they give the highest number of nodes in the table. Permutably homogeneous functions, on the other hand, 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. Other classes of functions such as linear, monotone, or symmetric functions lies between these two extremes.

The computation of the fitness function for the maximum separable subset is faster than its computation for the longest strip. This because, to find the maximum separable subset we need only to check two strips (the end strips), whereas we need to check all strips in order to obtain the longest strip generated by a weight vector. However for the same method (GA or

TB) and function, the solution using maximum separable subsets produces more nodes than the solution using longest strips. The reason is that removing the longest strip generated by a $\vec{w}$ may create completely new strips (it can happen only when the removed strip is not an end strip) which are the union of two strips that enclose the removed strip. For instance, consider three strips where one has value 1 and the other two have values 0 and suppose strip of value 1 is between strips of values 0, that is we have the sequence 010. Then removing strip 1 creates a new strip 00, which may be longer than the strips 1 and 0 together. And the longer are the created strips then the smaller will be the number of new added nodes. This situation cannot happen with solution using maximum separable subset. That is why the maximum separable solution creates more nodes than the longest strip solution.

In most of our experiments, GA methods give slightly better results than TB methods when K^n is the training set, and vice versa, TB methods give slightly better results than GA methods when 60% of K^n is the training set. It is interesting to note that functions which have larger separation than random functions may require more nodes for approximation (second part of table) than for exact identification (first part of table). The example of Figure 6.8 justifies the reason. Our constructive learning algorithm behaves as a *greedy algorithm*, that is it tries to obtain the maximum possible subset G at each local step (iteration) —so it performs a local optimization— in order to minimize the total number of nodes in the constructed network. However a sequence of local optimizations does not guarantee that the end-product (i.e the number of nodes) will be optimal. This is particularly true when only a fraction of the given function is given to the as training set. In Figure 6.8a), the algorithm will create exactly two nodes (for first layer) for learning the function, if it finds $\vec{w}_1$. In Figure 6.8b) if, by any chance, the algorithm finds $\vec{w}_2$ then there will be no way for him to create less than 3 nodes (for first layer) even though $\vec{w}_2$ is the best possible solution at the first iteration of the algorithm.

The same justification may also explain why tabu search performs poorly (but only slightly), in our experiments, than genetic search. Our TB has a better local optimization than our GA, and that is this *good* feature which prevents him from reaching the minimum number of nodes. In other words TB is more greedy than GA.

As a last observation, the constructed networks have a very low accuracy for functions with smaller separation, which is not a surprise since many points in the testing set will not fall within the regions covered by the neurons defined at the first layer. The large separation of the permutably

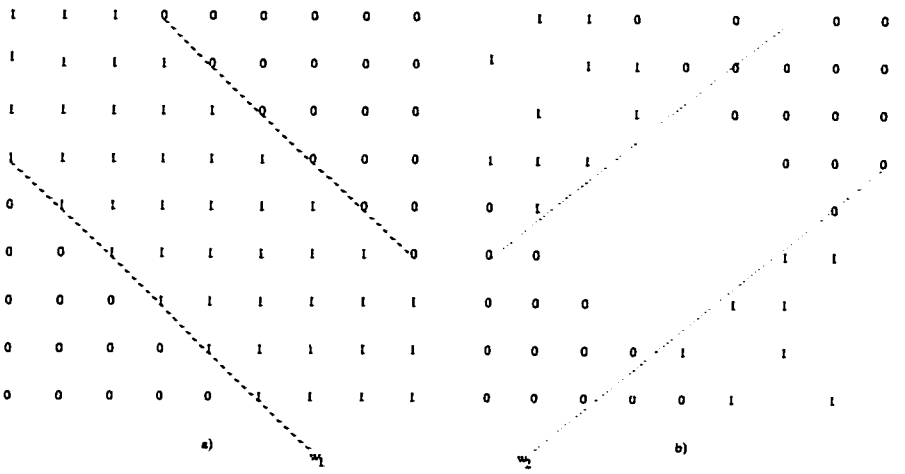


Figure 6.8: Function and its portion.

homogeneous functions makes their constructed networks to cover large areas. Therefore their networks accuracy is higher.

6.6 Conclusion

In this chapter 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 and tabu search to find a (near) minimal set of hidden units that partitions the space $V \subseteq K^n$. Our main contribution is indeed the development of growth algorithms for synthesizing multiple-valued logic functions by neural networks.

As part of a future research we are interested to study the generalization and clustering properties of such networks. An interesting application of our algorithm in the area of image processing is the following. Suppose a set of points of different colors are distributed on a two-dimensional plane. The problem is to partition the plane into regions by drawing a minimum number of infinite straight lines such that each region contains points of single color only (there may be more than one region for a particular color). Our algorithm can be modified to separate the colors in images.

Chapter 7

Minimization of Multiple-Valued Multiple-Threshold Perceptrons Using Genetic Algorithms

We address the problem of computing and learning multiple-valued multiple-threshold perceptrons. Every n-input k-valued logic function can be implemented using a (n, k, s)-perceptron, for some number of thresholds s. We propose a genetic algorithm to search for an optimal (n, 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.

7.1 Introduction

A problem still left open in the domain of MVMT functions is how to minimize the number of thresholds in order to construct the most efficient MVMT networks or units. To minimize the number of thresholds, traditional techniques of MVMT 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 (MVMT synthesis of *k-valued symmetric functions*, for instance). The MVMT 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 MVMT networks synthesis. Also, only some small classes of k-valued logic functions are considered for MVMT synthesis techniques.

Given $f \in P_k^n$ we are looking for 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 $F_{k,s}^n$. In this chapter, 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 $F_{k,s}^n(\vec{r}) = f$. We propose genetic algorithms as techniques for minimizing multivalued multithreshold perceptrons.

7.2 Computing minimal s-representations with genetic algorithms

For the s-representation problem, the search space is the space of weight vectors $\vec{w}$. Therefore we encode the possible solutions as vectors $\vec{w} \in R^n$ and design appropriate genetic operators which are suitable for the s-representation problem. The crossover and mutation operators used in the chapter are identical to those defined in chapter 6. Also the population is initialized in the same way as in chapter 6. So only the fitness function needs to be redefined for this problem.

7.2.1 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 $\vec{w}\vec{x}$ and construct a sorted array (or list) of records of the form $(\vec{w}\vec{x}, f(\vec{x}))$. The array is sorted using $\vec{w}\vec{x}$ as primary key and $f(\vec{x})$ as secondary key. Let these records be sorted as follows: $\vec{x}_1, \ldots, \vec{x}_{k^n}$, or more precisely, $(\vec{w}\vec{x}_i, f(\vec{x}_i)), 1 \leq i \leq k^n$, where $\vec{w}\vec{x}_i \leq \cdots \leq \vec{w}\vec{x}_{k^n}$. Then $\vec{w}\vec{x}_j$ is a threshold if $f(\vec{x}_{j-1}) \neq f(\vec{x}_j)$. 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 $(1 \leq i \leq T(\vec{w}) \leq$

$k^n - 1$) is *valid* if all points $\vec{x} \in K^n$ lying in its corresponding hyperplane H_i (given by $\vec{w}\vec{x} = t_i$) 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 $\vec{w}\vec{x}_1 = \vec{w}\vec{x}_2$ but $f(\vec{x}_1) \neq f(\vec{x}_2)$. 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

$$Fitness1(\vec{w}) = 1 - \frac{T(\vec{w})}{k^n - 1} \tag{7.1}$$

Notice that a GA always maximizes an objective function and since $1 \leq T(\vec{w}) \leq k^n - 1$, then $Fitness1(\vec{w})$ 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 7.1 we can alternatively use formula 7.2 below.

$$Fitness2(\vec{w}) = \frac{2 - \frac{T(\vec{w})}{k^n - 1} - \frac{I(\vec{w})}{T(\vec{w})}}{2} = 1 - \frac{T(\vec{w})}{2(k^n - 1)} - \frac{I(\vec{w})}{2T(\vec{w})} \tag{7.2}$$

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 $0 \leq I(\vec{w}) \leq T(\vec{w})$ and thus $Fitness2(\vec{w})$ 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 $I(\vec{w}) \neq 0$ 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 nk^n steps to compute all the $\vec{w}\vec{x}$'s, $k^n \log k^n$ steps to sort them and at most k^n steps to compute $T(\vec{w})$. Therefore the evaluation of $Fitness(\vec{w})$ has a time complexity of $O(nk^n \log k)$.

Also, crossover and mutation operations take $O(n)$ steps each and the initialization of the population takes $O(npk^n \log k)$ 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 (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 $O(ngpk^n \log k)$.

7.3 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 (than other possible values) 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 (and to reduce the disruptive effect of a high crossover rate). 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 (for reasons explained in section 7.2.1). Also we used *stochastic universal selection shceme* as our reproduction method.

It is interesting that the proposed population (chromosomes) 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 with small k versus large k and small n versus large n in order to see how the number of invalid thresholds changes. Such experiment were performed on random functions and their results are reported in tables 7.1 and 7.2.

As we can see in both tables (and also in Table 7.3), the number of invalid thresholds obtained is always zero. The last column in both tables shows the running time for $s = 100$ and $g = 100$. Although our method is slow, it is no surprise that the algorithm is slower as n grows than as k grows. Such results agree with the complexity analysis given in 7.2.1.

k	#Invalids	#Seconds
2	0	3.25
4	0	8.92
8	0	36.72
16	0	153.76
32	0	700.12
64	0	3289.59
128	0	14511.80
256	0	99434.36

Table 7.1: Results for some two-input k-valued random functions.

n	#Invalids	#Seconds
2	0	3.25
4	0	10.40
8	0	217.47
9	0	496.61
10	0	1047.64
11	0	2155.61
12	0	4797.34
13	0	9918.57
14	0	21959.14

Table 7.2: Results for some n-input two-valued random functions.

From neural networks applications perspective, results on the number of invalid thresholds for $k \leq 32$ and $k \geq 64$ given in Table 7.1 are interesting, since these values of k correspond to discretizations of *real-valued* neurons by at most 5 bits or at least 6 bits, for fixed n. A more theoretical, not well understood problem would address $\frac{n}{\log n}$, n, or constant number of bits since we know that $\frac{n}{\log n}$ bits is sufficient, n bits is the best known lower bound, while constant number of bits appears to be sufficient in practice for non-malicious threshold functions.

In Table 7.3 we show results of ten runs of the GA on examples of n-place k-valued logic functions (for $2 \leq k \leq 4$ and $2 \leq n \leq 7$) given by

$$f(\vec{x}) = \lfloor (\sum_{i=1}^{n} \frac{1}{a_i} x_i) + n \rfloor \bmod k \tag{7.3}$$

where $a_i = 2i + 1$. We can easily guess the minimum number of thresh-

olds needed for a perceptron to simulate these functions. Indeed, 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. In Table 7.4 we show examples of optimal solutions obtained by the GA.

n	Optimal s	Number of runs	Average number of generations
		k = 4	
2	1	10	0
3	2	10	24.6
4	2	9	430.9
		k = 3	
5	1	3	669
		k = 2	
6	1	9	283.78
7	1	3	660.33

Table 7.3: Some results for 10 runs.

n	$\vec{w}$	$\vec{t}$	$\vec{o}$
	k = 4		
2	0.953823 0.300371	2.508386	2 3
3	0.830144 0.473697 0.294061	2.303273 4.793706	3 0 1
4	0.785003 0.441867 0.347428 0.260417	2.351256 4.714851	0 1 2
	k = 3		
5	-0.754707 -0.465486 -0.348600 -0.228958 -0.199491	-2.285579	0 2
	k = 2		
6	0.487796 0.827506 0.205418 0.075150 0.111406 0.130513	1.595870	0 1
7	0.746777 0.459637 0.299486 0.148247 0.255323 0.191369 0.132577	1.654147	1 0

Table 7.4: Some optimal solutions found.

In Table 7.3, the second column indicates the optimal number of thresholds that the GA must find, the third columns contains the number of runs where the GA reached the optimum, and the fourth column shows the average number of generations over all successfull runs needed to obtain the

optimal solution. All solutions found by the GA, optimal or not, were valid solutions in that they do not contains invalids thresholds.

As seen from the table, 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 (meaning more genetic operations) to successfully obtain an optimum. This is indicated by the fourth column. For $k = 4$ and $n = 5$, for example, the GA could not find an optimum within five runs of 1000 generations each, however it was successfull within one run with 2000 generations. This suggest that given enough time (which depends on n) the GA will always find the minimal s-representation for a logic function f. We do not have rows for higher values of n because of the fact that the algorithm is slow as n grows.

It is interesting to note that the functions we used in our experiments are the most difficult for the GA since the their s-representations are very small (e.g. $s \in \{1, 2\}$). This indicates that for most (random) functions the GA will perform much better than for our test functions because s is larger on average.

We compared our technique with the *extended permutably homogeneous* (k, s)*-perceptron learning algorithm* (EPHPLA) described in [Ngom 98c]. A permutably homogeneous perceptron has a $(s+1, k)$-permutation as its output vector, that is a permutations of $s+1$ elements out of K with $s \leq k-1$. The EPHPLA generalizes the homogeneous $(k, k-1)$-perceptron learning algorithm of [Obradović 94] and has a time complexity of $O(enk^n)$, where e is the number of learning epochs. The EPHPLA can only learn permutably homogeneous functions and an example of such class of functions are our test functions given by equation (7.3). It is proven in [Ngom 98c] that the EPHPLA always converges for permutably homogeneous functions, and that also, it always finds a minimal s-representation for a learned function f. The EPHPLA is faster and outperform the GA on learning these same test functions within one run of 1000 learning epochs. The GA converged better only for $n = 2$ and any k. The main advantage of the GA method over the EPHPLA is that it can learn any logic function provided enough time is given.

7.4 Open problems and further research

One major problem with our method is that it is very slow for even small values of n ($n \geq 10$) and k ($k \geq 8$). Better techniques are needed for efficient

computation of (k, s)-perceptrons. For example, one possible solution could be to use a population of s-representations $\vec{r}$ (instead of weights $\vec{w}$) and then design a fitness function which minimizes s and minimizes the error between the (k, s)-perceptrón and a given multiple-valued logic function. Sorting is not needed here since we select only chromosomes whose $\vec{t}$ is sorted. Those with unsorted $\vec{t}$ will have severe penalty.

Another aspect that would be of interest to neural networks community is related to the proposed GA optimization. If it generates small number of invalid thresholds when working with K^n, it should also be able to discover a near optimal representation for smaller subsets of K^n. If so, and if we believe in *Occam's razor principle* (that the simplest satisfactory explanation of a phenomenon is most likely to be a correct one) then this is interesting from generalization perspective. In particular, instead of calculating $\vec{w}\vec{x}$ for every $\vec{x} \in K^n$, the same fitness estimation algorithm can be applied to a small enough subset $S \subset K^n$ (e.g. $(k-1)^n$ randomly selected points from K^n). Once a faithful representation for S (called the training set) is learned, it is easy (but interesting) to measure classification error on out of sample data (e.g. total number of mistakes, and maybe sum of squared errors on $K^n - S$). If applied for generalization and if needed, the fitness function can be further modified to allow some error if it results in smaller representation.

7.5 Conclusion

In this chapter, the optimization of multiple-valued multiple-threshold perceptrons using a genetic algorithm have been discussed. Experimental evidence show that the genetic search can be very effective however slow it may be. Better methods are needed in order to decrease the complexity of the objective function. The GA can possibly be useful in constructing efficient homogeneous multivalued multilayer neural networks. Finally, generalization properties of the GA can be studied when the fitness function is modified to work with small subsets of K^n.

Chapter 8

Neuro-Genetic Minimization of Canonical Multiple-Valued Logic Expressions

The only known algorithm for finding minimal multiple-valued logic expressions is exhaustive search. The excessive computation time makes this approach impractical. Especially, multiple-valued sum-of-products expressions (such as $\vee$-of-$\wedge$ expressions for instance) are interesting because of the ease with which they can be implemented by programmable logic arrays [Bender 85, Sasao 89, Sasao 86]. Because of the computational complexity associated with minimal sum-of-products solutions, there is considerable interest in heuristics. Typical heuristics is that, first a minterm is selected and then an implication is chosen that covers the minterm [Besslich 86, Dueck 92a, Fei 93, Miller 94b, Muroga 79, Muzio 86, Sasao 89, Tirumalai 91, Yang 90]. This process is repeated until the given expression is covered. [Dueck 92b, Yildirim 93, Yurchak 90] proposed multiple-valued logic design methods which employ *simulated annealing* [Kirkpatrick 83]. [Hata 94, Kaczmarek 95] proposed neural network techniques. [Hata 97, Miller 94a, Wang 96, Zaitseva 96] proposed solutions using genetic algorithms. [Kabakçioglu 90, Lloris-Ruiz 93, Roman-Roldan 92] used information theoretic approaches to minimization. Here we propose a new technique, a *neuro-genetic* minimization method which combine the powers of genetic algorithms and neural networks to search for a minimal circuit (or logic expression) that simulates a function as accurate as possible.

8.1 Introduction

We have seen in section 2.4 that, for a given multiple-valued logic function, a (near) minimal network must be realized first and then one can apply the back-propagation algorithm in order to find the window literal parameters. It is very difficult to obtain a minimal canonical representation for a given function. A canonical representation (V-of-$\wedge$ form) has at most k^n $\wedge$-terms each containing n literal operators and there exist no efficient method for minimizing such representation. Recall the formula of a canonical V-of-$\wedge$ representation form

$$f(\vec{x}) = \bigvee_{i=1}^{m} (c_i \wedge {}^{a_{i1}}x_1^{b_{i1}} \wedge \cdots \wedge {}^{a_{in}}x_n^{b_{in}}) \tag{8.1}$$

where c_i, a_{ij}, b_{ij} are constant over K, $a_{ij} \leq b_{ij}$, $1 \leq i \leq m \leq k^n$, $1 \leq j \leq n$, and $\vec{x} \in K^n$.

In this section we propose a genetic algorithm combined with a modified version of the backpropagation learning method described in [Tang 95] to search for a minimal representation (network) that efficiently computes a given multiple-valued logic function.

8.2 Problem representation

Our main problem is to minimize a (or find a minimal) multiple-valued logic V-of-$\wedge$ expression for computing a given but arbitrary logic function. We use two levels of representation of the problem: genetic and neural representations.

8.2.1 Genetic representation

In the first level of representation, the *genetic representation*, our problem is represented as a set of binary strings, that is a population of chromosomes. Each chromosome encodes a potential solution of the problem, that is a possible logic expression, as $M_1 M_2 \cdots M_m$ where M_i $(1 \leq i \leq m \leq k^n)$ corresponds to a $\wedge$-term of the solution. For simplicity of description, let k be a power of 2. Each M_i is a sequence $c_1^i c_2^i \cdots c_{\log k}^i l_1^i l_2^i \cdots l_n^i$ of $\log k + n$ bits, where $c_1^i c_2^i \cdots c_{\log k}^i$ is the binary representation of the constant in the i-th $\wedge$-term and

$$l_j^i = \begin{cases} 1 & \text{if the } i\text{-th } \wedge\text{-term contains the variable } x_j \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

where $1 \leq j \leq n$. For example, for $k = 4$ and $n = 4$ the $\vee$-of-$\wedge$ representation

$$\begin{aligned} f(x,y,z,t) &= (3 \wedge {}^{a}t^{b}) \vee (2 \wedge {}^{c}x^{d} \wedge {}^{e}z^{f}) \vee \\ &\quad (1 \wedge {}^{g}y^{h} \wedge {}^{i}z^{j} \wedge {}^{k}t^{l}) \vee (3 \wedge {}^{m}x^{n} \wedge {}^{o}y^{p}) \end{aligned} \tag{8.3}$$

corresponds to the chromosome $\overbrace{110001}^{M_1}\overbrace{101010}^{M_2}\overbrace{010111}^{M_3}\overbrace{111100}^{M_4}$.

The initial population will consist of chromosomes of variable lengths $(\log k + n)m$, where $1 \leq m \leq k^n$ is a random integer number.

[Hata 97] describes an interesting expression minimization method that uses genetic algorithm only. Its genetic representation, however, gives longer chromosomes than ours. It uses a complex binary encoding scheme in order to encode the minterms, the literal terms and their parameters. For instance, if a minterm M has the constant 3 in the logic expression (that is we have $f(\vec{x}) = \cdots + 3M + \cdots$) and B is the binary representation of M, then the chromosome c representing the expression $f(\vec{x})$ is of the form $c = \cdots BBB \cdots$. So every minterm with constant d is copied d times in the chromosome. B encodes whatever is in M, that is: the variables in M, the literal terms and the literal parameters. This binary representation of expressions yields lenghty chromosomes evaluations, crossovers and mutations and hence may not be realistic for even small values of k and n (e.g. $k = 8$ and $n = 2$). To reduce the length of representation we introduce the neural representation that encode the literal parameters. So we do not need to encode them in a chromosome.

8.2.2 Neural representation

Our chromosome representation, however, does not encodes the window literal parameters a's and b's of an associated expression. There are two alternatives in encoding the parameters. The first alternative is to include their binary representations in the chromosome. This, however, will make the decoding of the chromosome very difficult, meaning that we must find a way to know where in the chromosome are the parameters. Also this representation yields quite long chromosomes and lenghty running times.

The second alternative and better solution is to learn the literal parameters by a neural network learning algorithm. Therefore we need to represent a chromosome as a neural network which encodes the literal parameters. Since a chromosome encodes a canonical logic expression, then we need only to represent its associated expression as a depth-three multiple-valued logic network (circuit) of window literal gates, $\wedge$ gates and $\vee$ gates, and apply

a learning algorithm to learn the window literal parameters. For instance, Figure 8.1 shows the network corresponding to expression (8.3).

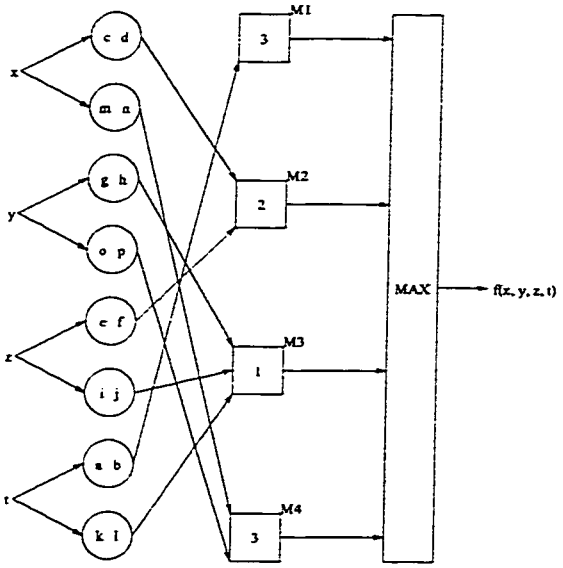


Figure 8.1: Multiple-valued logic network associated with (8.3).

So we have a four-layer network characterized as follows.

- Layer 0 is the input layer and contains n nodes called *input nodes.*
- Hidden layer 1 contains $U(c)$ nodes called *literal nodes.* The weight connection on the edge from an input node to a literal node is set to 1 if the variable associated with the input node corresponds to the literal associated with the literal node, otherwise the weight is 0. For instance, the edge from the input node associated with the variable x to the literal node associated with the literal ${}^{o}y^{p}$ is 0, whereas the weight from x to the literal node associated with ${}^{c}x^{d}$ is 1. In the figure we show only edges of weight 1 from input nodes to literal nodes. Each literal node has two thresholds t_1 and t_2 (with $t_1 \leq t_2$, and $t_1, t_2 \in [0, k-1]$), called *literal parameters*, and computes a simple function $F_L : K^n \mapsto K$ of the form

$$F_L(\vec{x}) = {}^{t_1}\vec{w}\vec{x}^{t_2} = \begin{cases} k-1 & \text{if } t_1 \leq \vec{w}\vec{x} \leq t_2 \\ 0 & \text{otherwise} \end{cases} \tag{8.4}$$

where $\vec{x} \in K^n$ is the input vector and $\vec{w} \in \{0,1\}^n$ is the weight vector associated with $\vec{x}$. The literal node's transfer function is a window literal function ${}^{t_1}y^{t_2} = \begin{cases} k-1 & \text{if } t_1 \leq y \leq t_2 \\ 0 & \text{otherwise} \end{cases}$, $y \in K$. Since there is exactly one input node corresponding to the literal node then there is exactly one edge with weight 1 entering the literal node (any other edge entering the literal node has weight 0). Suppose that the edge from the i-th input node of value x_i has weight 1 then we will have $\vec{w}\vec{x} = x_i$ since $w_j = 0$ for all $j \neq i$ and $w_i = 1$. Therefore the literal node selects only the input node to which it is associated.

- Hidden layer 2 contains $M(c)$ nodes called $\wedge$-*nodes*. Each $\wedge$-node corresponds to a $\wedge$-term of the associated canonical expression. A $\wedge$-node contains a threshold $t \in K$, which is simply the constant presents in its associated $\wedge$-term. The weight connection from a literal node to a $\wedge$-node is 0 if, in the canonical expression, the literal associated with the literal node is in the $\wedge$-term associated with the $\wedge$-node, otherwise the weight is set to $k-1$. For instance the edge connecting the literal node of thresholds a and b to $\wedge$-node M_1 has weight 0 since the literal ${}^at^b$ is in the first $\wedge$-term of the corresponding canonical expression, whereas the weight connection from the same literal node to $\wedge$-node M_4 is $k-1$ since ${}^at^b$ is not in the last $\wedge$-term of the corresponding expression. In the figure we show only edges of weight 0 from literal nodes to $\wedge$-nodes. Each $\wedge$-node computes a simple function which is the minimum between its threshold t and the minimum of the maximums between the inputs and there associated weights. That is

$$F_\wedge(\vec{x}) = t \wedge \bigwedge_{i=1}^{U(c)} (w_i \vee x_i) \tag{8.5}$$

where $\wedge$ (resp. $\vee$) is the multiple-valued logic *minimum* (resp. *maximum*) function which returns the minimum (resp. maximum) of its argument, $\vec{x} \in K^{U(c)}$ is the input to the $\wedge$-node and $\vec{w} \in \{0, k-1\}^{U(c)}$ is its associated weight vector. The $\vee$ operator here plays the role of an input selector. That is only the output from literal nodes which are relevant to the $\wedge$-node are selected for minimum. A literal node is relevant to the $\wedge$-node if its associated literal term is in the associated $\wedge$-term of the $\wedge$-node. Therefore for an irrelevant literal node L_i which outputs a value x_i, the weight from L_i to the $\wedge$-node is $w_i = k-1$, and since $w_i \vee x_i = k-1$ then the input x_i will not affects the outcome

of the minimum operation since $(w_i \vee x_i) \wedge y = (k-1) \wedge y = y$ for any y. On the other hand, if L_i is relevant to the $\wedge$-node then $w_i = 0$ and $w_i \vee x_i = x_i$, and therefore we have $(w_i \vee x_i) \wedge y = x_i \wedge y$ whose outcome depends on input x_i. Thus the $\wedge$-node selects only the inputs which are relevant to it.

- Layer 3 is the output layer and contains only one node called $\vee$-*node*. The weights from the $\wedge$-nodes to the $\vee$-node are all fixed to 1 and the $\vee$-node computes a simple function which is the maximum of its inputs, that is

$$F_{\vee}(\vec{x}) = \bigvee_{i=1}^{M(c)} (w_i x_i) = \bigvee_{i=1}^{M(c)} x_i \tag{8.6}$$

where $x_i \in K$ is the output of the i-th $\wedge$-node of layer 2.

In the network, the only learnable parameters are the literal parameters. Any other parameter such as weights on the edges or threshold at the $\wedge$-nodes are fixed from the beginning of the learning algorithm.

Our network topology is similar to the topology used in [Tang 95]. However, in [Tang 95] all the weights are fixed to 1 and the nodes compute the weighted sum of their inputs before applying their corresponding node functions (literal, minimum, or maximum functions). Our setting of weights enable the learning algorithm to update only the literal parameters that are relevant to a given $\wedge$-node. Later we will explain what we mean by that and why it is important.

The genetic and neural representations induce two levels of search. The genetic search (first level) that finds a minimal possible expression for a function, and, the neural search (second level) that finds the literal parameters of that possible expression such that the function is implemented by the expression.

8.3 Genetic search

Given a population of potential solutions representing $\vee$-of-$\wedge$ expressions, we search for a solution among this population that has the smallest possible length and number of variables, and which simulates the given logic function as accurate as possible. To efficiently perform this search we need to define an appropriate objective function and good search operators. These are described in the following sub-sections.

8.3.1 Fitness function

Consider a given chromosome c and the network associated with it. Our objective function is based on three criteria. We want to minimize i) the error between the network output and the desired output, ii) the number of $\wedge$-nodes in the network (i.e. the number of $\wedge$-terms in the canonical representation) and, iii) the number of window literal nodes in the network (i.e. the number of variables in the expression). Let $E(c)$ be the number of inputs vectors which are incorrectly classified by the network, $M(c)$ be the number of $\wedge$-terms in the chromosome, and $U(c)$ be the number of ones in chromosome c corresponding to the variables in the canonical representation (we exclude the ones corresponding to the constants). We propose the following fitness functions for a chromosome c.

$$\mathit{Fitness1}\,(c) = 4 - \frac{E(c)}{k^n} - \frac{U(c)}{nk^n} - \frac{M(c)}{k^n} - \frac{|E(c) - M(c)|}{k^n} \tag{8.7}$$

Since $0 \leq E(c) \leq k^n$, $1 \leq U(c) \leq nM(c) \leq nk^n$, $1 \leq M(c) \leq k^n$, and $0 \leq |E(c) - M(c)| \leq k^n$, then $\mathit{Fitness1}\,(c)$ and $\mathit{Fitness2}\,(c)$ (below) are maximal if and only if $E(c)$, $U(c)$, $M(c)$ and $|E(c) - M(c)|$ are all minimal.

The accuracy parameter $E(c)$ is more important than $U(c)$ and $M(c)$. That is we are more interested to obtain solution c with 100% accuracy, that is $E(c) = 0$. An alternative fitness function would thus be

$$\mathit{Fitness2}\,(c) = \frac{3 - \frac{U(c)}{nk^n} - \frac{M(c)}{k^n} - \frac{|E(c)-M(c)|}{k^n}}{1 + \frac{E(c)}{k^n}} \tag{8.8}$$

The term $\frac{|E(c)-M(c)|}{k^n}$ in both equations is very important, it makes $M(c)$ and $E(c)$ to move at the *same speed* toward zero. If the difference between them is large and that $M(c)$ is smaller than $E(c)$, the phenomenon that arises is that $E(c)$ cannot be minimized any more. The worst thing is that when $M(c)$ reaches a certain threshold toward zero then $E(c)$ starts to increase and moves toward k^n. This simply means that there is not enough $\wedge$-terms and literal variables in the associated formula for computing the given function. To avoid this we must prevent $M(c)$ to move faster than $E(c)$. That is there must always be a sufficient number of $\wedge$-terms in order to be able to minimize $E(c)$ while minimizing $U(c)$ and $M(c)$.

8.3.2 Crossover

We consider using five crossover operators. Let p_1 and p_2 be two parent chromosomes to be crossed over, then children c_1 and c_2 can be obtained by

applying one of the following five crossover operators to their parents (the operators are selected with equal probabilities). Let $M(p_1) \geq M(p_2)$.

Uniform crossover

We set $M(c_1) = M(p_1)$ and $M(c_2) = M(p_2)$. Then to obtain child c_i $(1 \leq i \leq 2)$ we randomly select a position j_i in p_1 such that $1 \leq j_i \leq M(p_1) - M(p_2) + 1$ and apply (8.9)

$$c_i = \begin{cases} p_1 & \text{if } random() \leq 0.5 \\ p_2 & \text{otherwise} \end{cases} \tag{8.9}$$

where $random()$ returns a random real number in the range [0, 1]. Figure 8.2 shows an example of uniform crossover.

p_1	=	110001	101010	010111	111100
p_2	=		010101	110010	
c_1	=	110001	110010	010010	111100

p_1	=	110001	101010	010111	111100
p_2	=	010101	110010		
c_2	=	110001	101010		

Figure 8.2: Uniform crossover with $M(p_1) = 4$, $M(p_2) = 2$, $j_1 = 2$ and $j_2 = 1$.

We can also apply uniform crossover at any bit position b, such that $1 \leq b \leq (\log k + n)(M(p_1) - M(p_2) + 1)$, of p_1 instead of at the $\wedge$-term positions of p_1 only.

Uniform crossover preserves the length of the parents. So using only uniform crossover is not good since if the optimal solution has a length which is not in the set of lengths of the population then it can never be reached. Therefore we need to design crossover operators which are not length-preserving.

One-point crossover

Here we randomly select positions j_1 and j_2 in p_1 such that $1 \leq j_1 < j_2 \leq M(p_1) - M(p_2) + 1$ and exchange the segments of p_1 and p_2 which are beyond the crossover point as shown in Figure 8.3.

p_1	=	110001	101010	010111	111100
p_2	=		010101	110010	
c_1	=	110001	101010	110010	
c_2	=	010101	010111	111100	

Figure 8.3: One-point crossover with $M(p_1) = 4$, $M(p_2) = 2$, $j_1 = 2$ and $j_2 = 3$.

Here also, we can apply one-point crossover at bit positions instead of at $\wedge$-term positions. One-point crossover is not length-preserving.

Shuffle-and-pick crossover

In this method, we put all $\wedge$-terms of both p_1 and p_2 into an urn. We first generate a random integer number $1 \leq M(c_1) \leq Min(M(p_1) + M(p_2), k^n) - 1$ and set $M(c_2) = Min(M(p_1) + M(p_2), k^n) - 1 - M(c_1)$, thus $M(c_1) + M(c_2) = M(p_1) + M(p_2)$. Then we randomly pick $M(c_1)$ $\wedge$-terms from the urn to form c_1 (by appending the $\wedge$-terms) and $M(c_2)$ $\wedge$-terms to form c_2. For example, Figure 8.4 shows the children obtained by shuffling the $\wedge$-terms of p_1 and p_2.

p_1	=	110001	101010	010111	111100	
p_2	=	010101	110010			
c_1	=	101010				
c_2	=	111100	110001	010111	110010	010101

Figure 8.4: Shuffle-and-pick crossover with $M(c_1) = 1$ and $M(c_2) = 5$.

Shuffle-and-pick crossover is not length-preserving.

Term exchange crossover

In m-term exchange crossover, we randomly choose an integer m such that $1 \leq m \leq M(p_2)$ and exchange exactly m $\wedge$-terms of p_1 with m $\wedge$-terms of p_2. In this method, smaller m yields more conservative children (that is the children are more similar to their parents) and thus less exploratory search (i.e. less diverse population), whereas larger m results to more dissimilar children and therefore a less tendency of the genetic algorithm to prematurely converge to local optima. As shown in Figure 8.5 we randomly pick m $\wedge$-terms of p_1 and exchange them to m random $\wedge$-terms of p_2.

p_1	=	110001	101010	010111	111100
p_2	=	010101	110010		
c_1	=	110001	101010	010111	010101
c_2	=	111100	110010		

Figure 8.5: m-term exchange crossover with $m = 1$.

Term exchange crossover is length-preserving.

Term migration crossover

We first randomly select a *source* parent, say p_1 for instance, and then randomly pick m $\wedge$-terms from the *source* parent such that $1 \leq m < M(source) - 1$. Child c_1 is the *source* parent minus the m selected $\wedge$-terms and child c_2 is the *destination* parent plus the m selected $\wedge$-terms. In other words, the m selected $\wedge$-terms migrate from the *source* parent to the *destination* parent (see Figure 8.6).

p_1	=	111100	110001	010111	110010	010101
p_2	=	010101	110010			
c_1	=	111100	010111	010101		
c_2	=	010101	110010	110001	110010	

Figure 8.6: m-term migration crossover with $source = p_1$ and $m = 2$.

Term migration crossover is not length-preserving. Here again, smaller m yields more conservative children than larger m.

One of the five crossover operators is selected with a probability of $\frac{1}{5}$ when two parents are chosen for crossover.

8.3.3 Mutation

We propose four techniques of mutation. The *bit mutation* in which we randomly alternate bits at some bit positions of a parent. The *term mutation* in which we randomly alternate all bits at some $\wedge$-term positions of a parent. The *m-term reduce mutation* in which m randomly selected terms of a parent are removed. The *m-term augment mutation* in which a random bit string of m terms is added to a parent. Figures 8.7-8.10, respectively shows examples of such mutations.

$$\begin{array}{lcl} p_1 & = & 110001 \quad 101010 \quad 010111 \quad 111100 \\ c_1 & = & 010001 \quad 100110 \quad 010110 \quad 101001 \end{array}$$

Figure 8.7: Bit mutation.

$$\begin{array}{lcl} p_1 & = & 110001 \quad 101010 \quad 010111 \quad 111100 \\ c_1 & = & 110001 \quad 010101 \quad 101000 \quad 111100 \end{array}$$

Figure 8.8: Term mutation.

$$\begin{array}{lcl} p_1 & = & 110001 \quad 101010 \quad 010111 \quad 111100 \\ c_1 & = & 110001 \quad 010111 \end{array}$$

Figure 8.9: m-term reduce mutation with $m = 2$.

During mutation of a chromosome, one of the four mutation operators is selected with a probability of $\frac{1}{4}$. The last two operators are not length-preserving.

$$\begin{array}{lcl} p_1 & = & 110001 \quad 101010 \\ c_1 & = & 110001 \quad 101010 \quad 101000 \quad 111100 \end{array}$$

Figure 8.10: m-term augment mutation with $m = 2$.

8.4 Neural search

The neural search is the process that searches for a set of literal parameters for a given candidate solution (obtained by the genetic search) such that the given but arbitrary function is computed as closely as possible. The method we use to learn the literal parameters of our network is the backpropagation learning algorithm, in which the parameters are updated so as to minimize the error between network's output and the desired output (that is the function's output). The error function to minimize is a sum of squared errors of the form

$$E = \frac{1}{2}\frac{1}{|P|}\sum_{p=1}^{|P|}(Max_p - t_p)^2 \tag{8.10}$$

where $P \subseteq K^n$ is the set of input vectors (i.e. the patterns to be trained, the training set), t_p is the desired output for the V-node when pattern $\vec{x}_p$ is applied to the input layer of the network (that is $f(\vec{x}_p) = t_p$), and Max_p is the actual output of the V-node for input vector $\vec{x}_p$.

Backpropagation employs gradient descent to minimize E. It updates the literal parameters a_i and b_i of literal nodes L_i in the network until E falls below a certain value. To do this, the partial derivative of E with respect to each literal parameter in the network is computed. This is the sum of partial derivatives for each of the patterns. That is, for a literal node L_i, its literal parameters a_i and b_i are updated as follows

$$a_i = a_i + \Delta a_i \text{ and } b_i = b_i + \Delta b_i \tag{8.11}$$

where

$$\Delta a_i = -\eta\frac{\partial E}{\partial a_i} \text{ and } \Delta b_i = -\eta\frac{\partial E}{\partial b_i} \tag{8.12}$$

and η $(0 < \eta \leq 1)$ is the learning rate and controls the speed of learning. Let $\vec{x} \in K^n$ be an input vector applied to the network. Let ${}^{a_i}x_i^{b_i}$ (for some

$1 \leq i \leq U(c)$) be the node function of the literal node L_i. Then using the chain rule gives

$$\begin{aligned}\frac{\partial E}{\partial a_i} &= \frac{\partial E}{\partial Max}\frac{\partial Max}{\partial a_i} \\ &= \frac{\partial E}{\partial Max}\frac{\partial Max}{\partial Min}\frac{\partial Min}{\partial a_i} \\ &= \frac{\partial E}{\partial Max}\frac{\partial Max}{\partial Min}\frac{\partial Min}{\partial ({}^{a_i}x_i{}^{b_i})}\frac{\partial ({}^{a_i}x_i{}^{b_i})}{\partial a_i}\end{aligned} \tag{8.13}$$

similarly we have

$$\frac{\partial E}{\partial b_i} = \frac{\partial E}{\partial Max}\frac{\partial Max}{\partial Min}\frac{\partial Min}{\partial ({}^{a_i}x_i{}^{b_i})}\frac{\partial ({}^{a_i}x_i{}^{b_i})}{\partial b_i} \tag{8.14}$$

where Min is the output of the $\wedge$-node to which node L_i is relevant, that is such that the edge connecting them has weight 0. It should be noted that for every literal node, there is exactly one edge of weight 0 going from that literal node to layer 3, the others outgoing edges from that node have weight $k-1$.

In order to do learning with gradient descent, derivatives of the literal function, $\wedge$ function and $\vee$ function are needed. Unfortunately none of them are differentiable.

Differentiating E with respect to Max for a particular pattern $\vec{x}$ yields

$$\frac{\partial E}{\partial Max} = t - Max \tag{8.15}$$

Let Min_m be the output of the m-th $\wedge$-node and let the literal node L_u be relevant to it, for some $1 \leq m \leq M(c)$ and $1 \leq u \leq U(c)$. We gives the following heuristic approximations for the derivatives of the $\vee$ and $\wedge$ functions, similar to those described in [Tang 95].

$$\frac{\partial Max}{\partial Min_m} = \begin{cases} 0.1 & \text{if } Min_m < \bigvee_{i \neq m} Min_i \\ 0.5 & \text{if } Min_m = \bigvee_{i \neq m} Min_i \\ 0.9 & \text{if } Min_m > \bigvee_{i \neq m} Min_i \end{cases} \tag{8.16}$$

and

$$\frac{\partial Min_m}{\partial ({}^{a_u}x_u{}^{b_u})} = \begin{cases} 0.9 & \text{if } {}^{a_u}x_u{}^{b_u} < t_m \wedge \bigwedge_{i \neq u} {}^{a_i}x_i{}^{b_i} \\ 0.5 & \text{if } {}^{a_u}x_u{}^{b_u} = t_m \wedge \bigwedge_{i \neq u} {}^{a_i}x_i{}^{b_i} \\ 0.1 & \text{if } {}^{a_u}x_u{}^{b_u} > t_m \wedge \bigwedge_{i \neq u} {}^{a_i}x_i{}^{b_i} \end{cases} \tag{8.17}$$

where t_m is the threshold value at the m-th $\wedge$-node. In equation (8.17) the minimum is over all $i \neq u$ such that node L_i is relevant to the $\wedge$-node node M_m.

For the literal function we use the same derivatives as [Tang 95]. They are given as

$$\frac{\partial({}^{a_u}x_u{}^{b_u})}{\partial a_u} = -1 \tag{8.18}$$

and

$$\frac{\partial({}^{a_u}x_u{}^{b_u})}{\partial b_u} = 1 \tag{8.19}$$

Our derivatives are directly derived from [Tang 95]. However, in [Tang 95] the minimum in equation (8.17) is over all $i \neq u$, $1 \leq i \leq U(c)$. Also, [Tang 95] uses 0 (resp. 1) instead of 0.1 (resp. 0.9) in both equations. The reason for these modifications is that, first, we want to ensure that a literal parameter is always updated if it is relevant to $\wedge$-node which has contributed to the error at the output. Having a 0 or 1 may not yields an update of a relevant literal. Second, for a $\wedge$-node that has contributed to the error at the output of the network, we do not want to update the literal parameters that are irrelevant to it. Clearly if the node M_2 in Figure 8.1 contributes to the error then we need only to update the parameters c, d, e and f, and it seems illogical to update other parameters such as a and b for instance. This also ensures that a literal node will not unlearn what it has already learned (memorized) unless its relevant $\wedge$-node contributes to the error. That is why we need to set the weights as described in section 8.2.2 in order to select only the literal nodes that are relevant to a given $\wedge$-node. In this way, training is faster since it focuses only on the nodes that contribute to the error, unlike conventional backpropagation networks where all parameters are updated.

Finally, the learning rules for updating the literal parameters a_i and b_i of literal node L_i are

$$\Delta a_i = -\eta(t - Max)\frac{\partial Max}{\partial Min}\frac{\partial Min}{\partial({}^{a_i}x_i{}^{b_i})} \tag{8.20}$$

and

$$\Delta b_i = +\eta(t - Max)\frac{\partial Max}{\partial Min}\frac{\partial Min}{\partial({}^{a_i}x_i{}^{b_i})} \tag{8.21}$$

where Min is the output of the $\wedge$-node relevant to L_i.

In [Tang 95] the following equation is given for parameter a_i (for instance)

$$\Delta a_i = -\eta(t - Max) \sum_{m=1}^{M(c)} \frac{\partial Max}{\partial Min_m} \frac{\partial Min_m}{\partial(^{a_i}x_i{}^{b_i})} \tag{8.22}$$

This equation gives slower learning time than ours because all parameters are updated, which also makes *all* literal nodes which do not participate to the error to *forget* what have been learned before (thus adding more learning time) when new erroneous examples are given to the network. Also it is not clear (or it is not said) in [Tang 95] how a given node in the network functions. We think that each node performs a weighted sum of its input before applying the node's function, that is literal, minimum or maximum function (depending to the type of node). The reason we think that is that all weights of the network are fixed to 1 and hence a node does not selects the inputs which are relevant to it. Also the weighted sum can be larger than $k-1$, meaning that they must do something else to keep the sum within interval $[0, k-1]$. Our setting of weights and the nodes' functions makes it simpler and faster to train the network.

Let $S \subseteq K^n$ be the set of input vectors. Each epoch of the back-propagation algorithm goes as follows. For every vector $\vec{x} \in S$ apply $\vec{x}$ to the input layer of the network and propagate it through the network in order to obtain its corresponding output at the $\vee$-node of the network. Then compute the error at the output, and bacpropagate the error through the network in order to compute how much each $\wedge$-node and literal node has contributed to the error. Finally, update each literal parameter according to equations (8.20) or (8.21). This process is continued until we reach a maximum number of epochs or the error E falls below an appreciable limit.

8.5 Experiments and discussions

To analyze the performance of our method, we generated random multiple-valued $\vee$-of-$\wedge$ expressions of random lengths in $\{\frac{k^n}{2}, k^n\}$ and use their associated functions as inputs to our neuro-genetic (NG) algorithm. So we expect at least that the NG algorithm will obtain the same generated expressions (functions) or, if we are lucky enough, to find a shorter representation of their respective functions.

We used a single run of the algorithm in each generated expression. The GA parameters are: 100 generations, population size of 50, crossover rate of 75% and mutation rate of 0.5%. The NN parameters are: 100 learning

epochs and learning rate of 10%. Our experiments for $k = 4$ and $n = 2$ are summerized in the nine tables given below. In the tables, we show the random multiple-valued logic expressions generated by the GA (under the label *Original Expression*) along with their shorter representations obtained by the NG algorithm (under the label *Approximation Expression*). We show the chromosomal representation of the original expression (see section 8.2) as a sequence of $\wedge$-terms of length $(\log 4 + 2)$ bits. Under each such $\wedge$-term we show its defined constant (that is the decimal value of the first $\log 4$ bits) and, in the next row, the list of literal parameters associated to each variable present in the $\wedge$-term. For the approximate expression, the literal parameters obtained by NG are not integer-valued, they are real numbers ranging from $[0, k-1]$. The next two rows for each expression give respectively the number of $\wedge$-terms ($M(c)$) and the number of variables ($U(c)$) contained in the expressions. The last row of each table displays the number of errors (that is $E(c)$) produced by the approximate expression.

Original Expression									
Canon	0001	0001	0011	0110	0111	0011	1101	0011	1101	1010
Const	0	0	0	1	1	0	3	0	3	2
Param	1 1	1 1	3 3,1 1	3 3	0 0,0 0	3 3,1 1	2 2	1 3,1 3	3 3	0 1
#Mins	10									
#Vars	21									
Approximate Expression										
Canon	1010	1010	1101							
Param	1.99 1.99	0.00 1.08	1.92 3.00							
#Mins	3									
#Vars	7									
#Errs	2									

Original Expression								
Canon	1110	1011	1001	1011	1010	0011	0101	0001
Const	3	2	2	2	2	0	1	0
Param	0 1	0 0,3 3	3 3	3 3,0 3	2 2	2 3,3 3	3 3	2 3
#Mins	8							
#Vars	18							
Approximate Expression								
Canon	0011	1110	1001	0110				
Param	3.00 3.00,0.00 3.00	0.00 2.00	0.00 3.00	2.00 3.00				
#Mins	4							
#Vars	9							
#Errs	4							

Original Expression													
Canon	1101	0010	0101	0110	1111	0111	1111	1011	1010	1001	1111	0001	1110	0001
Const	3	0	1	1	3	1	3	2	2	2	3	0	3	0
Param	1 3	0 1	1 3	2 2	1 1, 1 2	3 3,2 2	0 2,1 3	3 3,1 1	1 1	1 3	2 3,2 2	3 3	0 2	0 1
#Mins	16													
#Vars	40													
Approximate Expression														
Canon	1101	1010												
Param	0.10 2.90	0.00 3.00												
#Mins	2													
#Vars	5													
#Errs	2													

The last two columns of the original expression in the third table are as

follows:

1110	0011
3	0
1 2	0 0,3 3

Original Expression								
Canon	0110	0001	0111	1110	1010	1010	0011	1001	0110
Const	1	0	1	3	2	2	0	2	1
Param	0 0	1 3	0 2,3 3	0 0	3 3	0 3	3 3,3 3	2 2	0 2
#Mins	9								
#Vars	19								
Approximate Expression									
Canon	0010	1010	0010	1001					
Param	2.56 3.00	0.92 3.00	0.00 1.28	0.00 3.00					
#Mins	4								
#Vars	6								
#Errs	4								

Original Expression									
Canon	1110	0011	1011	0111	0110	0101	0011	1010	0111	1111
Const	3	0	2	1	1	1	0	2	1	3
Param	1 3	0 3,3 3	3 3,0 2	3 3,2 2	2 3	1 3	2 2,3 3	2 3	0 0,0 2	2 2,3 3
#Mins	10									
#Vars	26									
Approximate Expression										
Canon	1110	0101								
Param	1.00 3.00	2.00 3.00								
#Mins	2									
#Vars	5									
#Errs	2									

Original Expression											
Canon	1101	1111	0111	0110	1101	1001	1101	0001	1010	1001	0001	0110
Const	3	3	1	1	3	2	3	0	2	2	0	1
Param	2 2	3 3,3 3	1 2,1 1	2 2	1 2	3 3	1 3	3 3	2 3	2 2	0 2	2 3
#Mins	12											
#Vars	29											
Approximate Expression												
Canon	1101	0101										
Param	0.75 3.00	0.75 3.00										
#Mins	2											
#Vars	5											
#Errs	2											

Original Expression										
Canon	0101	1110	1110	1010	1101	0011	1010	1010	1011	0011
Const	1	3	3	2	3	0	2	2	2	0
Param	0 1	2 3	0 0	1 1	3 3	0 0,3 3	3 3	0 0	1 3,1 1	3 3,0 1
#Mins	10									
#Vars	24									
Approximate Expression										
Canon	1101	0001	1001							
Param	0.00 3.00	0.88 3.00	0.00 1.08							
#Mins	3									
#Vars	6									
#Errs	3									

As seen in the tables, all approximate expressions achieve a high accuracy of 81.88% $\pm$ 5.47%, a reduction of 8.1 $\pm$ 2.88 Λ-terms and a reduction of 11.8 $\pm$ 4.21 literal variables. For instance, in the third table, the original

Original Expression												
Canon	0001	0001	0011	1111	0011	0111	0101	0010	1001	1101	0110	0011
Const	0	0	0	3	0	1	1	0	2	3	1	0
Param	2 2	0 3	1 3,2 3	3 3,1 2	2 2,1 3	0 3,1 3	1 1	1 2	3 3	3 3	0 1	1 2,2 2
#Mins	10											
#Vars	25											

Approximate Expression			
Canon	1011	0110	1101
Param	0.00 2.10,2.90 3.00	0.00 2.18	2.82 3.00
#Mins	3		
#Vars	8		
#Errs	3		

Original Expression										
Canon	1001	1111	0101	1011	1011	0010	1011	0010	0101	1010
Const	2	3	1	2	2	0	2	0	1	2
Param	3 3	2 2 0 3	2 3	1 1 2 2	3 3 3 3	1 2	1 3 0 3	2 3	0 3	1 3
#Mins	10									
#Vars	23									

Approximate Expression				
Canon	1110	1111	1010	1001
Param	2.00 2.00	2.00 2.00,3.00 3.00	1.00 3.00	0.00 3.00
#Mins	4			
#Vars	11			
#Errs	3			

logic expression of 16 $\wedge$-terms has been successfully approximated by a logic expression containing only two $\wedge$-terms.

8.6 Conclusion

In this chapter we have discussed a new approach for minimization of multiple-valued logic $\vee$-of-$\wedge$'s expressions. We believe that our method can be modified to solve other types of sum-of-products expressions (such as expressions using *set literal* operators in place of window literal operators, or expressions using other kinds of sum or product operators such as *min* or *TSUM* gates) as long as the derivatives for these operators can be appropriately defined. Also such expressions must be well represented to allow a meaningful and efficient neuro-genetic search.

Experiments with randomly generated logic functions show our method to produce with high accuracy much smaller expressions than the original ones. One problem with the method is that we rarely achieve 100% accuracy. To improve the accuracy, we could use results obtained from our technique as starting point —that is as *good* initial solution— for a direct covering approach (or any other heuristic method). In this way, the heuristic method would attempt to cover part of the original expression that is not covered by the near-optimal expression produced by NG method, however using our near-optimal expression as initial candidate solution. Also, more studies need to be done in order to design better fitness functions that guarantee much higher accuracy than ours and an average size no worst than ours.

We tested only with randomly generated expressions because they happen to be more difficult to optimize in general. However it will be interesting to experiment with other classes of expressions and to analyze the results for each class.

Chapter 9

Conclusion and Future Directions

Neural net computing is parallel distributed computing with an ensemble of elemental processors interconnected in ways reminiscent of biological neural nets.

Although the idea of computing with arrays of interconnected elemental processors is certainly not new, there is currently a resurgence of interest in this area. This resurgence was initiated through the introduction of a few new algorithms which made it possible, perhaps for the first time, to implement and experiment with some simple realizations of such interconnected nets of elemental processors. Some imaginative demonstrations of such nets helped to activate further interest and fascination with these new developments.

At the present time there is widespread activity in this area with much interest in the potential use of neural networks in signal and image processing and so on. There is also interest in the possibility of using such nets as computer models for studying the functioning of neuro-biological nets.

This thesis addresses an intellectual issue which is neither of these categories but is more related to the former than to the latter.

The point is that since we can now implement neural nets which can actually perform certain basic information processing functions, it is of interest to see if we can fashion nets or systems of nets which can reproduce (i.e. mimic) certain trains of actions regularly performed by humans. The challenge is to be able to do this without an *oracle* within the net, that is without *devine revelation* or guidance, so to speak, at critical junctures. Positive results gained in activities of such nature would not be interpreted

to indicate that that is indeed how human biological nets do function but might serve to suggest preferences among various ways of thinking of processing in biological nets.

The issue we have addressed in this thesis is that of implementing multiple-valued logic systems in neural networks. Therre are many kinds of multiple-valued logic algebras such as *fuzzy logic*, *probabilistic logic*, logical calculus with *rough sets*, etc. However, for the present we have concentrated on one such logic system, that is the *classical multiple-valued logic* as defined in this thesis.

In particular we have discussed original models of multiple-valued neurons and multiple-valued neural networks and studied their learning and computing powers.

This dissertation has addressed the following problems.

1. *Computing capacity of (n, k, s)-perceptrons.* The capacity of a neuron is defined as the total number of functions it can simulate. We introduced the concept of multilinear partition of a point set $V \subset R^n$ and the concept of multilinear separability of a function $f : V \mapsto K = \{0, \ldots, k-1\}$. Based on well known relationships between linear partitions and minimal pairs, we derived formulae for the number of multilinear partitions of a point set in general position and of the set K^2. From the number of multilinear partitions of V, we have obtained results on the capacity of a single (n, k, s)-perceptron, respectively for $V \subset R^n$ in general position and for $V = K^2$. Finally, we have described a fast polynomial-time algorithm for counting the multilinear partitions of K^2.

2. *Constructing a minimal network for learning a given but arbitrary function.* We have considered the problem of synthesizing multiple-valued logic functions by minimal neural networks. A genetic algorithm which finds either the longest strip or the maximum separable subset for a subset in K^n, by using different fitness functions, is described. A strip contains points located between two parallel hyperplanes. Repeated application of the genetic algorithm partitions the space K^n into certain number of strips, each of them corresponding to a hidden unit. We have constructed two neural networks based on these hidden units and show that they correctly compute the given but arbitrary multiple-valued logic function. Preliminary experimental results are discussed and compared with other methods such as tabu search techniques.

3. *Learning algorithms for* (n, k, s)*-perceptrons.* Learning abilities of (n, k, s)-perceptrons are examined. The previously studied homogeneous $(n, k, k-1)$-perceptron learning algorithm have been generalized to the permutably homogeneous (n, k, s)-perceptron learning algorithm with guaranteed convergence property. A permutably homogeneous perceptron is a neuron whose output vector $\vec{o}$ is a permutation on K. We have obtained a powerful learning method that learns any permutably homogeneously separable k-valued function given as input.

4. *Minimizing a* (n, k, s)*-perceptron.* Every n-input k-valued logic function can be implemented using a (n, k, s)-perceptron, for some number of thresholds s. We proposed a genetic algorithm to search for a minimal (n, k, s)-perceptron that efficiently realizes a given but arbitray function, that is to minimize its number of thresholds.

5. *Finding a minimal expression for a k-valued logic function.* The only known algorithm for finding minimal multiple-valued logic expressions is exhaustive search. The excessive computation time makes this approach impractical. Especially, multiple-valued sum-of-products expressions (such as $\vee$-of-$\wedge$ expressions for instance) are interesting because of the ease with which they can be implemented by programmable logic arrays. Because of the computational complexity associated with minimal sum-of-products solutions, there is considerable interest in heuristics. We have proposed a new heuristic for minimization that combines the powers of neural network learning and genetic search.

Future research directions The followings are some interesting open problems related to my research in this area.

PAC-learnability of (n, k, s)*-perceptrons.* Another complexity measure of neural networks is the *Vapnik-Chervonenkis dimension.* It is defined as the maximum size of a training set $T \subseteq V$ such that a given network realizes all functions defined on T. If the (n, k, s)-perceptrons have a finite VC-dimension then they may be PAC-learnable. That is, there may exist an efficient (n, k, s)-perceptron learning algorithm which uses only a polynomial number of examples and that, with high probability (which can be made as high as desired), the algorithm outputs a good approximation of a given function within a desired degree of accuracy (which can be made as high as desired). The VC-dimension of (n, k, s)-perceptrons gives a valid lower bound on the VC-dimension of linear decision lists and other related learning architectures.

Computing capacity of partitioning architectures. The computing capacity of linear decision lists or any architecture obtained by a partitioning algorithm is not known and is still an open problem. The technique we have applied for deriving capacity results on (n, k, s)-perceptrons may be extended or improved to give results for partitioning architectures. The main question is: in how many ways can we partition a set with s hyperplanes not necessarily parallel (for a given s)? This question also motivates for the investigation of the VC-dimension and PAC-learnability of such structures.

Learning algorithm for (n, k, s)-perceptrons networks. Further research is needed to develop efficient learning methods for some identified k-valued neural networks.

Learning with high-order threshold functions. Suppose the following generalization of a threshold function: $g(P(x)) = 0$ if $P(x) < t$ and, $g(P(x)) = 1$ if $t \leq P(x)$, where $P(x)$ is a polynomial of degree $d \geq 0$ and t is a threshold level. We say that g is a threshold function of order d. For instance, the well-known linear threshold function is a threshold function of order 1 with $P(x) = \vec{w}\vec{x}$. Geometrically, an order d threshold function is a separating hyper-surface of degree d. For example, a linear threshold is a separating hyper-plane that can be expressed by a polynomial of degree 1. Very little results are known in neural networks literature on using hyper-surfaces as discriminant functions, such as quadratic surfaces (parabola, hyperbola, circle, ellipses, spheres, ...), cubic surfaces, or surfaces of degree $d \geq 4$. It may be that hyper-surfaces are better separators than linear surfaces (i.e. hyper-planes). More studies need to be done.

Multilayer (n, k, s)-perceptrons. An interesting research problem is to develop learning algorithm for multilayer neural networks whose processing units are (n, k, s)-perceptrons. Such networks would have the ability to learn any k-valued function. The neural networks we have considered in this thesis are heteregeneous networks, meaning that the (n, k, s)-perceptron elements are not all identical (e.g. the number of thresholds or inputs are not all the same). Designing learning algorithms for homogeneous networks seems (to us) much easier than for heteregeneous networks. However for homogeneous networks, the problem we must solve first is to define a good differentiable error function in order to do learning with gradient descent similar to the backpropagation learning method.

Bibliography

[Abd-El-Barr 86a] M.H. Abd-El-Barr (1986), *On the design of multi-valued CCD logic circuits*, Ph.D. Thesis, University of Toronto.

[Abd-El-Barr 90a] M.H. Abd-El-Barr and H. Choy (1990), *On the synthesis of MVMT functions for PLA implementation using CCDs*, Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic, pp.316-323.

[Abd-El-Barr 91] M.H. Abd-El-Barr, H. Choy, A.K. Jain and R.J. Bolton (1991), *A comparative study of programmable realization techniques of multi-valued multi-threshold functions*, Proceedings of the 21st IEEE International Symposium on Multiple-Valued Logic, pp.372-381.

[Abd-El-Barr 89] M.H. Abd-El-Barr, T.D. Hoang and Z.G. Vranesic (1989), *Programmable realization of multi-valued multi-threshold functions*, Proceedings of the 19th IEEE International Symposium on Multiple-Valued Logic, pp.42-53.

[Abd-El-Barr 90b] M.H. Abd-El-Barr and Z.G. Vranesic (1990), *Cost reduction in the CCD realization of MVMT functions*, IEEE Transactions on Computing, vol.39.

[Abd-El-Barr 90c] M.H. Abd-El-Barr, Z.G. Vranesic and S.G. Zaki (1990), *New algorithmic synthesis techniques for multi-valued logic functions*, IEEE Transactions on Computing.

[Abd-El-Barr 85] M.H. Abd-El-Barr, Z.G. Vranesic and S.G. Zaki (1985), *Realization of multi-valued multi-threshold logic functions using CCDs*, Proceedings of the 15th IEEE Inter-

national Symposium on Multiple-Valued Logic, pp.126-136.

[Abd-El-Barr 86b] M.H. Abd-El-Barr, S.G. Zaky and Z.G. Vranesic (1986), *Synthesis of multivalued multithreshold functions for CCD implementation*, IEEE Transactions on Computing, V.C-35, N.2, February, pp.124-133.

[Abdel-Hamid 94] G.H. Abdel-Hamid and M.H. Abd-El-Barr (1994), *Decomposition-based synthesis of multiple-valued functions for threshold logic network realization*, Proceedings of the 24th IEEE International Symposium on Multiple-Valued Logic, pp.58-64.

[Acketa 91] D. Acketa and Joviša Žunić (1991), *On the number of linear partitions of the (m,n)-grid*, Information Processing Letters, North-Holland, V.38, pp.163-168.

[Aibara 72] T. Aibara and M. Akagi (1972), *Enumeration of ternary threshold functions of three variable*, IEEE Transactions on Computers, April 1972, pp.402-407.

[Anthony 92] M. Anthony and N. Biggs (1992), *Computational learning theory*, Cambridge University Press, New York.

[Baum 89] E.B. Baum (1989), *A proposal for more powerful learning algorithms*, Neural Computation 1, pp.201-207.

[Becker 94] B. Becker and R. Drechsler (1994), *Efficient graph based representation of multi-valued functions with an application to genetic algorithms*, Proceedings of the 24th IEEE International Symposium on Multiple-Valued Logic, pp.65-72.

[Bender 85] E.A. Bender, J.T. Butler and H.G. Kerkhoff (1985), *Comparing the SUM with the MAX for use in four-valued logic PLAs*, Proceedings of the 15th IEEE International Symposium on Multiple-Valued Logic, pp.30-35.

[Besslich 86] P.W. Besslich (1986), *Heuristic minimization of MVL functions: A direct cover approach*, IEEE Transactions on Computers, February, pp.134-144.

[Blum 88] A. Blum and R.L. Rivest (1988), Proceedings of the 1st Workshop on Computational Learning Theory, Morgan Kaufmann, pp.9.

[Cao 93] Q. Cao, O. Ishizuka, Z. Tang and H. Matsumoto (1993), *Algorithm and implementation of a learning multiple-valued logic network*, Proceedings of the 23rd IEEE International Symposium on Multiple-Valued Logic, pp.202-207.

[Chan 89] S.C. Chan, L.S. Hsu, S. Broody and H.H. Teh (1989), *Neural three-valued logic networks*, Abstract, Proc. Int'l Joint Conf. Neural Net., Washington DC, June, 18-22, vol.2, pp.594.

[Chan 88] 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.

[Cover 65] T.M. Cover (1965), *Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition*, IEEE Transactions on Electr. Comp., V.EC-14, pp.326-334.

[Denker 87] J. Denker, D. Schwartz, B. Wittner, S. Solla, J. Hopfield, R. Howard and L. Jackel (1987), Complex Systems, V.1, pp.877.

[Dietterich 95] , T.G. Dietterich and G. Bakiri (1995), *Solving multiclass learning problems via error-correcting output codes*, Journal of Artificial Intelligence Research, V.2, pp.263-286.

[Duda 73] R.O Duda and P.E. Hart (1973), *Pattern classification and scene analysis*, Wiley, New York.

[Dueck 92a] G. Dueck (1992), *Direct cover MVL minimization with cost-tables*, Proceedings of the 22nd IEEE International Symposium on Multiple-Valued Logic, pp.58-65.

[Dueck 94] G.W. Dueck and J.T, Butler (1994), *Multiple-valued logic operations with universal literals*, Proceedings of the 24th IEEE International Symposium on Multiple-Valued Logic, pp.73-79.

[Dueck 92b] G.W. Dueck, R.C. Earle, P. Tirumalai and J.T. Butler (1992), *Multiple-valued logic array minimization by simulated annealing*, Proceedings of the 22nd IEEE International Symposium on Multiple-Valued Logic, pp.66-74.

[Dueck 87] G.W. Dueck and D.M. Miller (1987), *A direct cover MVL minimization using the truncated sum*, Proceedings of the 17th IEEE International Symposium on Multiple-Valued Logic, pp.221-227.

[Druzeta 74] A. Druzeta, Z.G. Vranesic and A.S. Sedra, 1974, *Application of multithreshold elements in the realization of many-valued logic networks*, IEEE Trans.Comput., November, vol.C-23, pp.1194-1198.

[Edelsbrunner 87] H. Edelsbrunner (1987), *Algorithms in combinatorial geometry*, Springer Verlag, Heidelberg.

[Epstein 74] G. Epstein, G. Frieder and D.C. Rine (1974), *The development of multiple-valued logic as related to computer science*, Computer, V.7, pp.20-32.

[Fahlman 91] S.E. Fahlman and C. Lebiere (1991), *The cascade-correlation learning architecture*, Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, August.

[Fei 93] B. Fei, Q. Hong, H. Wu, M. Perkowski and N. Zhuang (1993), *Efficient computation for ternary Reed-Muller expansion under fixed polarities*, International Journal of Electronics, V.75, N.4, pp.685-688.

[Fleisher 87] M. Fleisher (1987), *The Hopfield model with multilevel neurons*, Proceedings of the IEEE Conference on Neural Information Processing Systems, Denver, CO, pp.278-289.

[Frean 90] M. Frean (1990), *The upstart algorithm: a method for constructing and training feedforward neural networks*, Neural Computation, 2, pp.198-209.

[Gallant 90] S.I. Gallant (1990), *Perceptron-based learning algorithms*, IEEE Transactions on Neural Networks, V.1, pp.179-191.

[Gallant 86] S.I. Gallant (1986), *Three constructive algorithms for network learning*, Proceedings of the 8th Annual Conferenceon Cognitive Science Soc., Amherst, Ma, August 15-17, pp.652-660.

[Glover 93] F. Glover, E. Taillard and D. De Werra (1993), *A User's Guide to Tabu Search*, Annales of Operating Research, V.41, pp.3-28.

[Goldberg 89] D.E. Goldberg (1989), *Genetic algorithms in search, optimization, and machine learning*, Reading, MA, Addison-Wesley.

[Golea 90] M. Golea and M. Marchand (1990), *A growth algorithm for neural network decision trees*, Europhysics Letters, V.12, N.3, pp.205-210.

[Gruau 92] F. Gruau (1992), *Genetic synthesis of Boolean neural networks with a cell rewriting developmental process*, COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks, L.D. Whitley and J.D. Shaffer eds., IEEE Computer Society Press.

[Hampel 71] D. Hampel and R.O. Winder (1971), *Threshold logic*, IEEE Spectrum, May, pp.32-39.

[Hanyu 89] T. Hanyu and T. Higuchi (1989), *High-density quaternary logic array chip for knowledge information processing systems*, IEEE Journal of Solid-State Circuits, SC-24, N.4, pp.916-921.

[Hardy 54] G.H. Hardy and E.M. Wright (1979), *An introduction to the theory of numbers, 5th edition*, Oxford, England, Clarendon Press.

[Haring 65] D. Haring (1965), *Multithreshold threshold elements*, IEEE Transactions on Electronic and Computer, V.EC-15, June, pp.45-65.

[Hata 97] Y. Hata, K. Hayase, T. Hozumi, N. Kamiura and K. Yamato (1997), *Multiple-valued logic minimization by genetic algorithms*, Proceedings of the 27th IEEE International Symposium on Multiple-Valued Logic, pp.97-102.

[Hata 94] Y. Hata, T. Hozumi and K. Yamato (1994), *System design based on neural computing*, Proceedings of the 3rd International Conference for System Integration, pp.197-205.

[Hata 93] Y. Hata, T. Hozumi and K. Yamato (1993), *Gate model networks for minimization of multiple-valued logic functions*, Proceedings of the 23rd IEEE International Symposium on Multiple-Valued Logic, pp.29-34.

[Hata 92] Y. Hata, F. Miyawaki and K. Yamato (1992), *Optimal output assignment and the maximum number of implicants needed to cover the multiple-valued logic functions*, Proceedings of the 22nd IEEE International Symposium on Multiple-Valued Logic, pp.389-395.

[Holland 75] J.H. Holland (1975), *Adaptation in natural and artificial systems*, Ann Arbor, MI, Michigan University Press.

[Hozumi 95] T. Hozumi, N. Kamiura, Y. Hata and K Yamato (1995), *Multiple-valued logic design using multiple-valued EXOR*, Proceedings of the 25th IEEE International Symposium on Multiple-Valued Logic, pp.290-295.

[Hurst 84] S.L. Hurst (1984), *Multiple-valued logic - its status and its future*, IEEE Transactions on Computer, V.C-33, N.12, pp.1160-1179.

[Imme 85] M. Imme and C. Papachristou (1985), *Simplification of MVL functions and implementation via a VLSI array structure*, Proceedings of the 15th IEEE International Symposium on Multiple-Valued Logic, pp.242-248.

[Ishizuka 79] O. Ishizuka, 1979, *Synthesis of MVMT networks for applying I^2L*, Proc. 9th IEEE ISMVL, pp.67-73.

[Ishizuka 77] O. Ishizuka (1977), *On MVMT networks composed of conventional threshold elements*, IEEE Transactions on Computing, V.C-26, N.2, December, pp.1251-1257.

[Ishizuka 76] O. Ishizuka (1976), *Multivalued multithreshold networks*, Proceedings of the 6th IEEE International Symposium on Multiple-Valued Logic, pp.44-47.

[Izumi 92] T. Izumi, S. Wakabayashi and R. Dang (1992), *A method for simplifying logic expression using Karnaugh map and neural network*, IEICE, Technical Report, VLD92-26, pp.49-54.

[Joo 92] N. Joo, K. Tomeoka, N. Muranaka and S. Imanishi (1992), *Prime implicant loop recognitions on a Karnaugh map for logical function by neural networks*, IEICE, Spring National Concention Record, D-207.

[Judd 87] S. Judd (1987), Proceedings of the IEEE 1st Conference on Neural Networks, San Diego, V.2, pp.685.

[Kaczmarek 95] A. Kaczmarek, V. Antonenko, S. Yanushkevich and E.N. Zaitseva (1995), *Algorithm for network to realize linear MVL functions using arithmetical logic*, Proceedings of the 12th International Conference on System Science, pp.23-30.

[Kabakçioglu 90] A.M. Kabakçioglu, P.K. Varshney and C.R.P. Hartmann (1990), *Application of information theory to switching function minimization*, IEE Proceedings, E, V.137, pp.389-393.

[Kameyama 87] M. Kameyama, T. Hanyu and T. Higuchi (1987), *Design and implementation of quaternary NMOS integrated circuits for pipeline image processing*, IEEE Journal of Solid-State Circuits, SC-12, N.1, pp.20-27.

[Keibek 92] S.A.J. Keibek, G.T. Barkema, H.M.A. Andree, M.H.F. Saveije and A. Taal (1992), *A fast partitioning algorithm and a comparison of feedforward neural networks*, Europhysics Letters, V.18, pp.555-559.

[Kerkhoff 84] H.G. Kerkhoff, 1984, *Theory, design and applications of digital charge-coupled devices*, Ph.D. Thesis, University of Twente, Netherlands.

[Kerkhoff 81] H.G. Kerkhoff and H.A. Robroek, 1981, *Multiple-valued logic CCDs*, IEEE Trans. Comput., September, vol.C-30, pp.644-652.

[Kerkhoff 80] H.G. Kerkhoff and H.A. Robroek, 1980, *The logic design of multivalued logic functions using multiple-valued CCDs*, Proc. 10th IEEE ISMVL, pp.146-151.

[Kirkpatrick 83] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi (1983), *Optimization by simulated annealing*, Science, V.220, N.4598, May, pp.671-680.

[Koplowitz 88] J. Koplowitz, M. Lindenbaum and A. Bruckstein (1990), *The number of digital straight lines on an $n \times n$ grid*, IEEE Transactions on Information Theory, V.36, N.1, January, pp.192-197.

[Krueger 86] F.R. Krueger (1986), *Comments on Takiyama's analysis of the multithreshold threshold element*, IEEE Transactions on Pattern Analysis and Machine Intelligence, V.PAMI-8, November, pp.760-761.

[Lim 92] J.H. Lim, H.C. Lui and H.H. Teh (1992), *A deductive neural logic system*, Proceedings of the 22nd IEEE International Symposium on Multiple-Valued Logic, pp.96-102.

[Littlestone 89] N. Littlestone (1989), *Mistake bounds and logarithmic linear threshold learning algorithms*, Technical Report CRL-89-11, University of California at Santa Cruz.

[Littlestone 87] N. Littlestone (1987), *Learning quickly when irrelevant attributes abound: A new linear threshold algorithm*, Machine Learning, V.1, N.2, pp.285-318.

[Lloris-Ruiz 93] A. Lloris-Ruiz, J.F. Gomez-Lopera and R. Roman-Roldan (1993), *Entropic minimization of multiple-valued functions*, Proceedings of the 23rd IEEE International Symposium on Multiple-Valued Logic, pp.24-28.

[Lu 85] H. Lu and S.C. Lee (1985), *M-algebra*, Proceedings of the 15th IEEE International Symposium on Multiple-Valued Logic, pp.272-285.

[Lukasiewicz 20] J. Lukasiewicz (1920), *O logice trojwartosciowej*, Ruch Filozoficzny, 15, pp.169-171.

[Marchand 93a] M. Marchand and M. Golea (1993), *On learning simple neural concepts: from halfspace intersections to neural decisions lists*, Network: Computation in Neural Systems, V.4, pp.67-85.

[Marchand 93b] M. Marchand and M. Golea (1993), *An approximation algorithm to find the largest linearly separable subset of training examples*, World Congress on Neural Networks: Proceedings of the Annual Meeting of the International Neural Network Society, Hillsdale, N.J., Erlbaum Associates, V.3, pp.556-559.

[Marchand 90] M. Marchand, M. Golea and P. Ruján (1990), *A convergence theorem for sequential learning in two-layer perceptrons*, Europhysics Letters, V.11, N.6, pp.487-492.

[McCulloch 43] W.S. McCulloch and W. Pitts (1943), *A logical calculus of ideas immanent in nervous activity*, Bullettin of Mathematical Biophysics, V.5, pp.115-133.

[Mezard 89] M. Mezard and J. Nadal (1989), *Learning in feedforward layered networks: the tiling algorithm*, Journal of Physics A, V.22, N.12, pp.2191-2203.

[Miller 94a] J.F. Miller, H.L. Luchian, P.V.G. Bradbeer and P.J. Barclay (1994), *Using a genetic algorithm for optimizing fixed polarity Reed-Muller expansions of Boolean functions*, International Journal of Electronics, V.76, N.4, pp.601-609.

[Miller 94b] J.F. Miller and P. Thomson (1994), *Highly efficient exhaustive search algorithm for optimizing canonical ternary Reed-Muller expansions of Boolean functions*, International Journal on Electronics, V.76, No.4, pp.37-56.

[Minsky 69] M. Minsky and S. Papert (1969), *Perceptrons: An introduction to computational geometry*, Cambridge, MA: MIT Press, Expanded edition 1988.

[Moody 88] J. Moody and C. Darken (1988), *Learning with localized receptive fields*, Proceedings of the 1988 Connectionist Models Summer School, pp.133-143.

[Muroga 79] S. Muroga (1979), *Logic design and switching theory*, John Wiley and Sons Inc..

[Muroga 71] S. Muroga (1971), *Threshold logic and its applications*, Wiley Interscience, New York.

[Muzio 86] J.C. Muzio and T.C. Wesselkamper (1986), *Multiple-valued switching theory*, Adam Hilger Ltd..

[Nadal 89] J. Nadal (1989), *Study of a growth algorithm for neural networks*, International Journal of Neural Systems, V.1, pp.55-59.

[Ngom 99a] A. Ngom (1999), *Neuro-genetic minimization of multiple-valued logic sum-of-product expressions*, Multiple-Valued Logic — An International Journal, to be submitted.

[Ngom 98a] A. Ngom (1998), *Genetic algorithm for the jump number scheduling problem*,Order — A Journal on the Theory of Ordered Sets and its Applications, V.14, July.

[Ngom 95a] A. Ngom (1995), *Set logic foundation of carrier computing*, Master Thesis, Computer Science Department, University of Ottawa, Ottawa, January, 113 pages.

[Ngom 99b] A. Ngom, V. Milutinović and I. Stojmenović (1999), *Synthesis of multiple-valued logic functions based on the longest strip or the maximum separable subset*, IEEE Transactions on Computers, to be submitted.

[Ngom 98b] A. Ngom, Z. Obradović and I. Stojmenović (1998), *Minimization of multiple-valued multiple-threshold perceptrons by genetic algorithms*, Proceedings of the 28th IEEE International Symposium on Multiple-Valued Logic, pp.209-214.

[Ngom 98c] 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, pp.161-166.

[Ngom 97a] A. Ngom, C. Reischer, D.A. Simovici and I. Stojmenovic (1997), *Set-Valued logic algebra: A carrier computing foundation*, Multiple-Valued Logic — An International Journal, V.2, N.3, pp.183-216.

[Ngom 97b] A. Ngom, C. Reischer, D.A. Simovici and I. Stojmenovic (1997), *Completeness criteria in set-valued logic under compositions with union and intersection*, Proceedings of the 27th IEEE International Symposium on Multiple-Valued Logic, pp.75-82.

[Ngom 95b] A. Ngom, C. Reischer and I. Stojmenovic (1995), *Classification of functions and enumerations of bases of set logic under boolean compositions*, Proceedings of the 25th IEEE International Symposium on Multiple-Valued Logic, pp.78-85.

[Ngom 99c] A. Ngom and I. Stojmenović (1999), *On the cardinality of maximal clones in set-valued logic containing union and intersection*, in progress.

[Ngom 99d] A. Ngom, I. Stojmenović and R. Tosić (1999), *On the number of linear partitions of multi-sets and the computing capacity of multiple-valued one-threshold perceptrons*, Proceedings of the 29th IEEE International Symposium on Multiple-Valued Logic, submitted.

[Ngom 99e] A. Ngom, I. Stojmenović and J. Zunić (1999), *On the number of multilinear partitions and the computing capacity of multiple-valued multiple-threshold perceptrons*, Proceedings of the 29th IEEE International Symposium on Multiple-Valued Logic, submitted.

[Nilsson 68] J. Nilsson (1968), *Learning machines. Foundations of trainable pattern classifying systems*, New York, McGraw-Hill.

[Nilsson 62] N.J. Nilsson (1962), *Learning machines*, McGraw-Hill, New York.

[Novikoff 62] A. Novikoff (1962), *On convergence proofs for perceptrons*, Proceedings of the Symposium on Mathematical Theory of Automata, New York, pp.615-622.

[Obradović 96] Z. Obradović (1996), *Computing with nonmonotone multivalued neurons*, Multiple-Valued Logic - An International Journal, V.1, N.4, pp.271-284.

[Obradović 94] Z. Obradović and I. Parberry (1994), *Learning with discrete multivalued neurons*, Journal of Computer and System Sciences, V.49, N.2, pp.375-390.

[Obradović 92] Z. Obradović and I. Parberry (1992), *Computing with discrete multivalued neurons*, Journal of Computer and System Sciences, V.45, N.3, pp.471-492.

[Obradović 90a] Z. Obradović and I. Parberry (1990), *Learning with discrete multivalued neurons*, Proceedings of the Seventh International Conference on Machine Learning, Austin, TX, pp.392-399.

[Obradović 90b] Z. Obradović and I. Parberry (1990), *Analog neural networks of limited precision I: Computing with multilinear threshold functions*, Advances in Neural Information Processing Systems II, D.S. Touretzky, ed., Morgan-Kaufmann, San Mateo, CA, pp.702-709.

[Obradović 90c] Z. Obradović and P. Yan (1990), *Small depth polynomial size neural networks*, Neural Computation, V.2, pp.402-404.

[Obradović 90d] Z. Obradović and P. Yan (1990), *Lower bounds for limited precision analog neural networks*, Technical Report CS-90-28, Department of Computer Sciences, Pennsylvania State University.

[Olafsson 88] S. Olafsson and Y.A. Abu-Mostafa (1988), *The capacity of multilevel threshold functions*, IEEE Transactions on Pattern Analysis and Machine Intelligence, V.10, N.2, pp.277-281.

[Parberry 90] I. Parberry (1990), *A primer on the complexity theory of neural networks*, Sourcebook of Formal Methods in Artificial Intelligence, ed. R. Banerji, North-Holland, pp.217-268.

[Parberry 88] I. Parberry and G. Schnitger (1988), *Parallel computation with threshold functions*, Journal of Computing and System Science, V.36, N.3, pp.278-302.

[Pomper 81] G. Pomper and J.A. Armstrong (1981), *Representation of multivalued functions using the direct cover method*, IEEE Transactions on Computing, September, pp.674-679.

[Post 21] E.L. Post (1921), *Introduction to a general theory of elementary propositions*, American Journal of Mathematic, 43, pp.163-185.

[Rivest 87] R.L. Rivest (1987), *Linear decision list*, Machine Learning, V.2, pp.229-246.

[Roman-Roldan 92] R. Roman-Roldan, J.F. Gomez-Lopera and A. Lloris-Ruiz (1992), *Multiple-valued function minimization by information theoretic methods*, Proceedings of the 1992 Conference on Information Sciences and Systems, Princeton, March 18-20, pp.454-459.

[Ruján 89] P. Ruján and M. Marchand (1989), *A geometric approach to learning in neural networks*, Complex System, V.3, pp.229-242.

[Rumelhart 86] D.E. Rumelhart, G.E. Hinton and R.J. Williams (1986), *Learning internal representations by error propagation*, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume I Foundations, D.E. Rumelhart, J.L. McClelland and the PDP Research Group, eds. MIT Press, Cambridge.

[Sasao 89] T. Sasao (1989), *On the optimal design of multiple-valued PLAs*, IEEE Computer, V.C-38, N.4, pp.582-592.

[Sasao 88] T. Sasao (1988), *Multiple-valued logic and optimization of programmable logic arrays*, IEEE Computer, April, pp.71-80.

[Sasao 86] T. Sasao (1986), *Programmable logic arrays: How to use and how to make*, (in Japanese) Tokyo, Japan, Nikkan Kougyou, May. IEEE Computer, April, pp.71-80.

[Schapire 97] R.E. Schapire (1997), *Using output codes to boost multiclass learning problems*, Proceedings of the 14th International Conference on Machine Learning, pp.313-321.

[Sethi 90] I.K. Sethi (1990), *Entropy nets: from decision trees to neural networks*, Proceedings of IEEE, V.78, pp.1605-1613.

[Sirat 90] J.A. Sirat and J.P. Nadal (1990), *Neural trees: a new tool for classification*, Network, V.1, pp.423-438.

[Siu 95] K.-Y. Siu, V. Roychowdhury and T. Kailath (1995), *Discrete neural computation: A theoretical foundation*, Prentice Hall Information and System Sciences Series, Thomas Kailath, Series editor, pp.68-103.

[Smith 88] K.C. Smith (1988), *A multiple-valued logic: A tutorial and appreciation*, Computer, V.21, N.4, pp.17-27.

[Smith 81] K.C. Smith (1981), *The prospects for multiple-valued logic: A technology and application view*, IEEE Transactions on Computer, V.C-30, pp.619-634.

[Takiyama 85] R. Takiyama (1985), *The separating capacity of a multithreshold threshold element*, IEEE Transactions on Pattern Analysis and Machine Intelligence, V.PAMI-7, January, pp.112-116.

[Takiyama 78] R. Takiyama (1978), *Multiple threshold perceptron*, Pattern Recognition, V.10, pp.27-30.

[Tan 96] C.L. Tan, T.S. Quah and H.H. Teh (1996), *An artificial neural network that models human decision making*, IEEE Computer, vol.29, N.3, March, pp.64-70.

[Tang 93] Z. Tang, O. Ishizuka, Q. Cao and H Matsumoto (1993), *Algebraic properties of a learning multiple-valued logic network*, Proc. 23rd IEEE International Symposium on Multiple-Valued Logic, pp.196-201.

[Tang 95] Z. Tang, O. Ishizuka and K. Tanno (1995), *Learning multiple-valued logic networks based on back-propagation*, Proceedings of the 25th IEEE International Symposium on Multiple-Valued Logic, pp.270-275.

[Tang 89] Z. Tang and Z. Li (1989), *T-multivalued algebra and circuits*, Chinese Journal of Computer, 4, pp.295-297.

[Teh 90a] H.H. Teh, L.S. Hsu, S.C. Chan and K.F. Loe (1990), *Temporal neural logic networks*, Proceedings of the 2nd International IEEE Conference of Tools for Artificial Intelligence, November 6-9, Herndon, VA, USA, pp.372-376.

[Teh 90b] H.H. Teh, L.S. Hsu, S.C. Chan and K.F. Loe (1990), *Multi-valued neural logic networks*, Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic, pp.426-436.

[Teh 90c] H.H. Teh, L.S. Hsu, S.C. Chan and K.F. Loe (1990), *Fuzzy neual logic networks*, Proceedings of the 23rd Hawaii International Conference on System Science, January 3-6, pp.363-368.

[Teh 89] H.H. Teh, L.S. Hsu, S.C. Chan and K.F. Loe (1989), *Probabilistic neural logic networks*, Proceedings of the Inter-Faculty Seminar on Neuronet Computing, June, National University of Singapor, Republic of Singapor, pp.76-93.

[Tirumalai 91] P.P. Tirumalai and J.T. Butler (1991), *Minimization algorithm for multiple-valued programmable logic arrays*, IEEE Transactions on Computers, February, pp.167-177.

[Tirumalai 84] P.P Tirumalai and J.T. Butler, 1984, *On the realization of multi-valued logic functions using CCD PLAs*, Proc. 14th IEEE ISMVL, pp.33-42.

[Tou 74] J.T. Tou and R.C. Gonzalez (1974), *Pattern recognition principles*, Addison-Wesley, Reading, MA.

[Tošić 92] R. Tošić (1992), *On the number of linear partitions*, Review of Research, Mathematical Series, V.22, N.1, Faculty of Science, University of Novi Sad, pp.141-149.

[Trotter 92] W.T. Trotter (1992), *Combinatorics and partially ordered sets - Dimension theory*, John Hopkins University Press, pp.215-217.

[Turán 96] G. Turán and F. Vatan (1996), *Linear decision lists and partitioning algorithms for the construction of neural networks*, Private communication.

[Vranesic 70] Z.G. Vranesic, E.S. Lee and K.C. Smith (1970), *A many-valued algebra for switching systems*, IEEE Transactions on Computer, C-19, pp.964-971.

[Wang 96] W. Wang and C. Moraga (1996), *Design of multiple-valued circuit using genetic algorithms*, Proceedings of the 26th IEEE International Symposium on Multiple-Valued Logic, pp.216-221.

[Watanabe 91] T. Watanabe et al. (1991), *Application of three-layered MVL neural networks to character recognition*, Inst. of Elect. Information and Comm. Engineers, Japan, Tech. Rep., vol.MVL-91, N.1, April, pp.45-52 (Japanese).

[Watanabe] T. Watanabe et al., *A layered neural network model using logic neurons*, Proceedings of the International Conference on Fuzzy Logic & Neural Networks, July, pp.675-678.

[Watanabe 92] T. Watanabe and M Matsumoto (1992), *Layered MVL neural networks capable of recognizing translated characters*, Proceedings of the 22nd IEEE ISMVL, pp.88-95.

[Watanabe 90] 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.

[Whitley 92] L.D. Whitley and J.D. Schaffer (1992), *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, IEEE Computer Society Press.

[Widrow 88] B. Widrow and R. Winter (1988), *Neural nets for adaptive filtering and adaptive pattern recognition*, IEEE Computer, V.21, N.3, March, pp.25-39.

[Yang 90] C. Yang and Y.M. Wang (1990), *A neighborhood decoupling algorithm for truncated sum minimization*, Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic, pp.153-160.

[Yanushkevich 95] S. Yanushkevich (1995), *Neural technologies and genetic algorithms in modern logic design*, Proceedings of the International Conference on Computer-Aided Design of Discrete Devices, vol.2.

[Yildirim 93] C. Yildirim, J.T. Butler and C. Yang (1993), *Multiple-valued PLA minimization by concurrent multiple and mixed simulated annealing*, Proceedings of the 23rd IEEE International Symposium on Multiple-Valued Logic, pp.17-23.

[Yuminaka 96] Y. Yuminaka, Y. Sasaki, T. Aoki and T. Higuchi (1996), *Wave-parallel computing technique for neural networks based on amplitude-modulated waves*, Proceedings of the 26th IEEE International Symposium on Multiple-Valued Logic, pp.210-215.

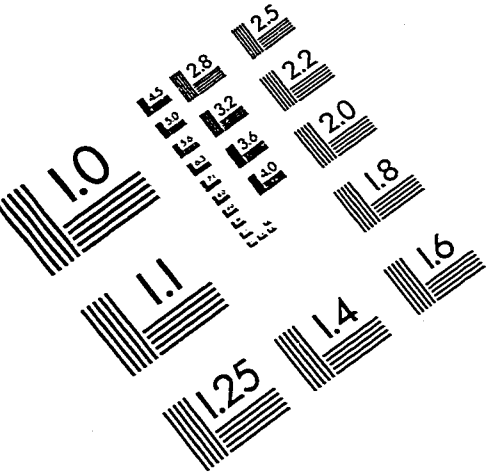
[Yurchak 90] J.M. Yurchak and J.T. Butler (1990), *HAMLET — An expression compiler-optimizer for the implementation of heuristics to minimize multiple-valued PLAs*, Proceedings of the 20th IEEE International Symposium on Multiple-Valued Logic, pp.144-152.

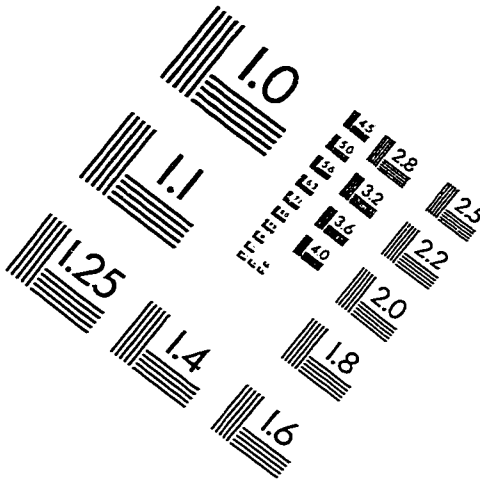
[Zaitseva 96] E.N. Zaitseva, T.G. Kalganova and E.G. Kochergov (1996), *Logical not polynomial forms to represent multiple-valued functions*, Proceedings of the 26th IEEE International Symposium on Multiple-Valued Logic, pp.302-307.

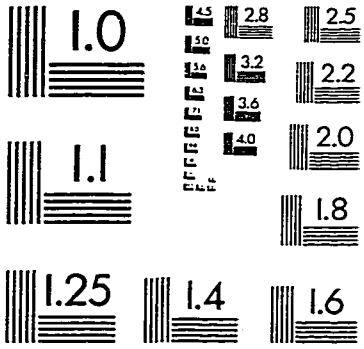
[Zaitseva 95] E.N. Zaitseva, E.G. Kochergov, A.A. Sanko and T.G. Kalganova (1995), *Genetic algorithm to optimize analytical description of multiple-valued functions*, Proceedings of the International Conference on Computer-Aided Design of Discrete Devices, V.2.

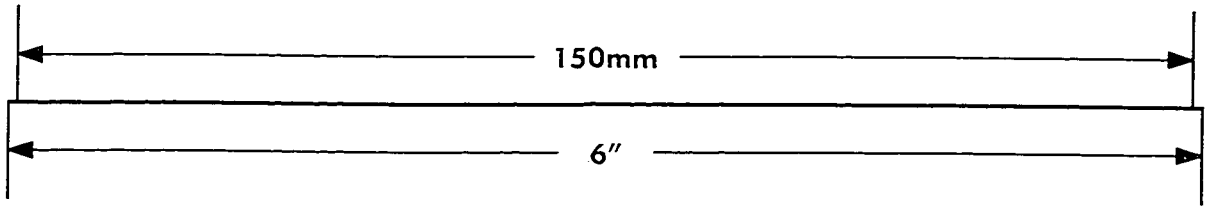
[Žunić 91] J. Žunić (1991), *On the asymptotic number of linear grid square partitions*, Bild und Ton.

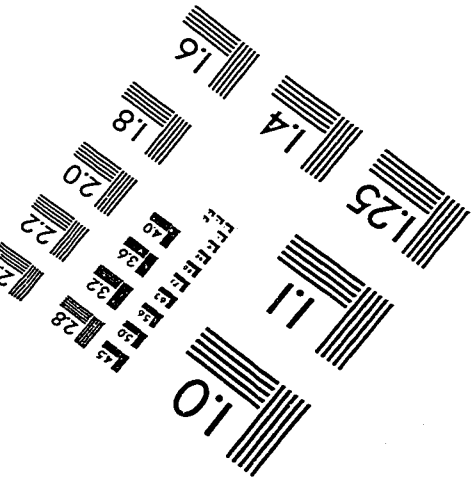
IMAGE EVALUATION TEST TARGET (QA-3)

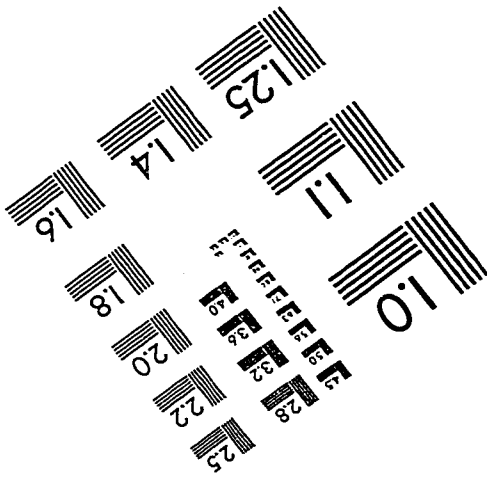












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989