ASSIGNMENT OF FUNCTIONAL RESPONSIBILITY IN PERCEPTRONS

Rik Achiel Verstraete

May 1986
CSD-860090

The dissertation of Rik Achiel Verstraete is approved.

E C Carterette

Edward C. Carterette

Allen Klinger

Tomás Lang

J Schlag

John D. Schlag

Jacques J. Vidal, Committee Chair

University of California, Los Angeles

1986

UNIVERSITY OF CALIFORNIA

Los Angeles

Assignment of Functional Responsibility in Perceptrons

A dissertation submitted in partial satisfaction of the
requirement for the degree Doctor of Philosophy
in Computer Science

by

Rik Achiel Verstraete

1986

LIST OF FIGURES

page

The dissertation of Rik Achiel Verstraete is approved.

E C Carterette

Edward C. Carterette

Allen Klinger

Tomás Lang

J Schlag

John D. Schlag

Jacques J. Vidal, Committee Chair

University of California, Los Angeles

1986

I owe a lot to my colleagues in our research group. I thank Tony Martinez and Antoine Cornuejols for patiently reading and commenting an early draft of my dissertation. I enjoyed working and meeting with them over the years, and I thank them for their contributions. Gabriella Adler worked with me on some detailed issues regarding functional decomposition in perceptrons; I owe some insight to her. In my work, I interacted with many other people of our group: I thank Paul Salas, Savio Chau, Mark Juckler, and Peter Berke.

I am also extremely grateful to Dr. Gerald Estrin, Chairman of the Computer Science Department at UCLA, for his stimulation and many constructive comments. Additionally, I thank all the graduate students in the CS202 *Advanced Computer Science Seminar;* without their endless stream of questions during my presentations I would have been unable to focus my work in as clear a fashion as I have now.

It may seem a formality, but I also thank my parents. They shaped my life and are directly responsible for my education and interests. Without them this work would not have existed.

Major financial support for my graduate study has been provided by the *I.B.M. T.J. Watson Research Center* in Yorktown Heights, NY, in the form of a Graduate Fellowship, and by *Aerojet ElectroSystems Corporation* in Azusa, CA, in the form of a Research Assistantship. I thank both corporations for funding my academic research.

ABSTRACT OF THE DISSERTATION

Assignment of Functional Responsibility in Perceptrons

by

Rik Achiel Verstraete
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1986
Professor Jacques J. Vidal, Chair

Perceptrons are defined to be multilayered networks of fixed topology that consist of adjustable combinational nodes. These nodes are not necessarily threshold gates, and the functional flexibility is present in all the layers (not just a single node or a single layer). The resulting network implements a modifiable Boolean function. The fundamental problem addressed is thus assignment of functional responsibility in a multilayered network.

The original perceptron research is reviewed and its extensiveness is demonstrated. Other existing implementations, both analog and digital, are also presented, and an implementation developed by us is discussed at

LIST OF TABLES

page

CHAPTER 1

INTRODUCTION

Perceptrons originated around 1958 in the work of Rosenblatt [Rose58]. He proposed a network of threshold gates as a model of pattern recognition in nervous systems. The topology of the proposed network is fixed, but the weights and threshold of the gates are modifiable. This flexibility can be used for adaptation and learning. Perceptrons are thus parallel and adaptive pattern recognition devices.

Later, Minsky and Papert formally proved several limitations of a simple perceptron structure [Mins69]. Their work, however, did not address the question of adaptive parallel processing in more general perceptrons.

The concept of adaptation in a network of adjustable threshold gates has received continued attention over the years. Many researchers, up to the present time, have shown possibilities for learning in specific classes of perceptron networks, but this work is not extensible to alternative networks [Bart81a, Bart82a, Fuku75, Fuku80, Fuku84, Miya84, Koho82, Taki78, Taki81].

This dissertation examines the perceptron in its fullest generality. For instance, the node of a perceptron is not restricted to a threshold gate, but can

ACKNOWLEDGEMENTS

It is a pleasant duty to begin this dissertation by acknowledging those who contributed to it.

First of all, I thank my advisor, Dr. Jacques J. Vidal, without whom I would not have been at UCLA. I thank him for his guidance, advice, and support; for carefully reading the draft of this text and giving so many suggestions for improvement; and for his inexhaustible enthusiasm for any work in the area of *neuromimetics*.

Much of the work reported here resulted from interactions I had with Don Moore. From the beginning to the end, he encouraged and helped me in many different ways. I thank him for an infinite number of discussions and for providing so many ideas.

I am also grateful to the members of my committee, who showed considerable interest in my work. I thank Dr. Allen Klinger, who taught me the principles of pattern recognition and perceptrons. I thank Dr. Tomás Lang, who read and commented the draft of my dissertation. I thank Dr. Edward C. Carterette and Dr. John D. Schlag, who evaluated my work from their point of view, respectively psychology or biology.

I thank Dr. Judea Pearl, who wanted to be on the committee but was unable to, for his enjoyable support over the years.

1.1.1. Parallel and Sequential Computing

Consider the decomposition of a computation into 4 subcomputations (figure 1.1). The size and complexity of these subcomputations depend on the particular level of interest. For instance, they could be Boolean functions of up to 3 inputs, AND gates or OR gates of a certain maximum fan-in, a 1-bit numerical operation, etc.

The steps 2, 3, and 4 in this figure are independent of each other and can execute in parallel. On the other hand, step 1 takes the outputs of steps 2, 3, and 4 as its inputs and therefore it can execute only when these three steps have completed their computations. Hence, steps 2 and 1 are sequential computation steps. The determining factor in parallel and sequential computation is the interdependence of data: parallelism requires independence; dependence implies sequential execution.

When a computation is broken into parts that are executed one after the other, the computation is said to be implemented in a sequential fashion. Sequential computing consists of a series of computational steps that must be executed in a proper order. The cause of the sequential nature of a computation can be either extrinsic or intrinsic to the problem itself.

In some problems, the external inputs are presented over discrete time steps and these problems must be implemented in a sequential fashion. The computational system then relies on internal memory to interpret the relationship between the data at different time steps. A detailed treatment of such *extrinsically* sequential problems can be found in textbooks on

VITA

May 18, 1958	Born, Lendelede, Belgium.
July 15, 1981	Burgerlijk Ingenieur Computerwetenschappen, Katholieke Universiteit Leuven, Leuven, Belgium.
December 1982	M.S. in Computer Science, University of California, Los Angeles, California.
1981-1983	Teaching Assistant, University of California, Los Angeles, California.

1.1.2. Perceptrons as Parallel Pattern Recognition Devices

Perceptrons implement an n-input dual pattern classification, assigning each input vector to a class C_0 or C_1 according to the value 0 or 1 of its output. This pattern classification is implemented as a network that has both width and depth. Each node of the network implements an atomic binary classification (a Boolean function). Figure 1.1 illustrates a possible structure of a small perceptron.

In the literature, perceptrons are usually associated with linear discrimination. The operation of a threshold gate, the basic building block of the early perceptrons, is indeed to weigh and sum its inputs and compare the result with a threshold value. The power of the perceptron idea, however, lies not in this weight-and-threshold operation, but in the possible use of *networks* of such units.

The multilayeredness is an essential characteristic of a perceptron. Yet, width in parallel computations is much more tractable than depth and most studies of computation therefore concentrate on width. A perceptron, on the other hand, is a simple model of parallel computation in which processing is distributed in a multilayered fashion.

Throughout this dissertation we will use Boolean algebra as our basic tool. Boolean logic is implementation independent and is the basic formalism for any digital computation. Furthermore, Boolean functions represent an asynchronous data flow model of computation (see for instance [Ager82, Kabl83]) and represent an extreme form of concurrent information

length.

Perceptrons are useful in applications that require a fast and modifiable implementation of Boolean functions. An important emerging application domain is presented: combinational rule-based systems. It is shown that some propositional-logic rule bases can be transformed into a pair of Boolean functions, which could be implemented with perceptrons.

Next, one aspect of the responsibility assignment in multilayered systems, namely the decomposition of a Boolean function on a given perceptron, is treated in detail. The theory of decomposition of Boolean functions is applied to this problem. The solution to the decomposition problem can be obtained in a straightforward fashion if the network has no fan-out connections. In a more general case, the decomposition problem is solved with a bottom-up search algorithm. At each node a few assignments are selected by a local selection criterion and tried in sequence. A reduction step between two nodes coordinates assignments to different nodes.

Finally, the requirement that the given network function be specified completely in advance is relaxed. Two approaches towards distributed learning by example in binary tree networks, taken from the literature, are reviewed, followed by our contribution. The theory of decomposition of Boolean functions is again the basic tool in this study.

require them to exhibit adaptation and learning.

In the remainder of this section we give a short review of adaptation in general, and how it is treated in the perceptron research and in our dissertation.

1.2.1. General Notions of Adaptation and Learning

The field of machine learning is still young and immature, and there is no commonly accepted definition of the terms *adaptation* or *learning*. In the literature one may find ''gradual improvement of system behavior as a result of past experience,'' or a similar definition. Adaptation is not a well defined capability, but includes a wide spectrum of interpretations, and it is not clear where *programming* stops and where *learning* begins.

The simplest form of adaptation is the ability to absorb and follow instructions. This assumes an *a priori* specification of a task as a set of complete instructions. A deduction process decomposes this task into a form suitable for the internal structure of the computational system. The complexity of this decomposition depends on the format of the instructions, the internal structure of the system, and the specifics of the task itself.

A more powerful form of learning is obtained when the task is specified incrementally, for instance as a series of examples provided by a teacher. A deduction process is then needed to adjust the internal computations to match the changing specifications. The system starts with incomplete knowledge and it must adapt to ever more precise specifications.

be any functionally adjustable gate. Furthermore, the functional flexibility of a perceptron is not restricted to a single node or a single layer of the network, but it is a property of all the layers in the network. By taking a generic point of view, we show that parallel processing and adaptation in perceptrons are broader issues than what is typically perceived by others.

This chapter introduces the three concepts related to the perceptron research, namely parallel pattern recognition (section 1.1), adaptation and learning (section 1.2), and neuromimetics (section 1.3). An overview of the remainder of the dissertation is given in section 1.4.

1.1. PARALLEL PATTERN RECOGNITION

The increasing demand for fast information processing is pushing the capabilities of sequential computers to a limit. In domains such as pattern recognition or artificial intelligence, the performance of even the fastest processors is already less than adequate. Furthermore, the demands are increasing at a faster rate than the improvements in speed of the components. With the price of hardware decreasing rapidly, parallel processing becomes more attractive.

The first section below reviews some notions of parallel and sequential computing. The next section then describes how perceptrons address the topic of parallel pattern classification.

1.3. NEUROMIMETICS

A large part of the work related to perceptrons has been motivated not only by the necessity to build faster and more adaptive machines, but also by the desire to build computer systems that embody characteristics found in animal brains. The emerging field of *neuromimetics* borrows some observations from neurophysiology and brain theory in the search for improved computer systems. These observations are of course not used literally, but are guiding principles. The premise is that the structure and operation of intelligent machines will necessarily mirror those of animal brains.

The perceptron model postulates that extreme fine-grain concurrent computing, combined with distributed learning, is a neuromimetic concept. Each node of a perceptron, combinational and adjustable, is a simplified model of the nerve cell. A parallel asynchronous flow of data through a network of combinational adaptive units is a closer model of the brain than the classical computer concept of centralized memory and a single sequential processor, the so-called von Neumann concept [von58]. The processing of data is fast and asynchronous, but the adaptive changes can occur at a slower pace. Furthermore, there is no explicit memory or state associated with the data processing. The state of the system consists of the *function* executed by each node. It is in other words *active memory*.

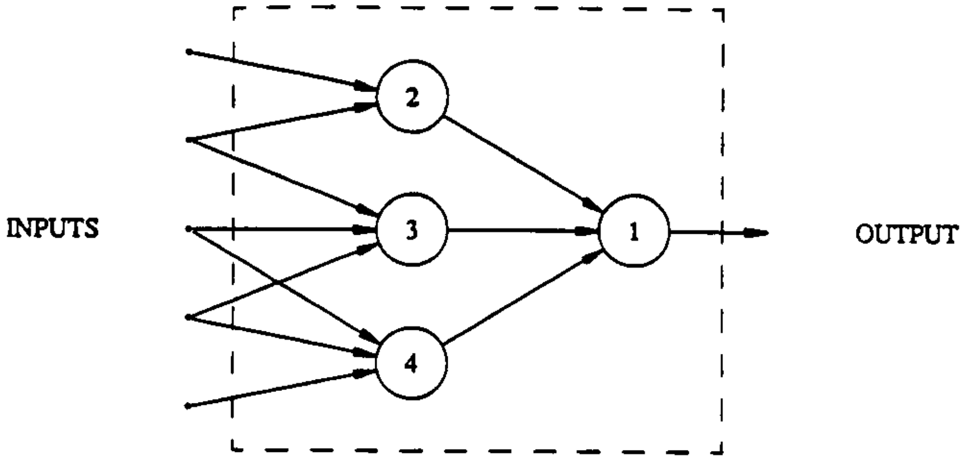


Figure 1.1: Schematic representation of a computation that is decomposed into 4 subcomputations.

automata theory [Hopc79, Lewi81].

Another form of sequential constraint on computing is *intrinsic:* the subcomputations at one level depend on results at the previous level (for instance the computations 2 and 1 in figure 1.1). This sequential constraint is not prescribed by external requirements, but is entirely intrinsic to the way the computation proceeds. This type of constraint creates *depth* of a computation.

In parallel computing, on the other hand, subcomputations execute concurrently and independently of each other. The number of independent subcomputations at any level corresponds to the *width* of the computation. Since the results of these parallel subcomputations must be subsequently combined, width and depth are always both present [Pate76].

given perceptron. We apply the theory of decomposition of Boolean functions to this problem and explain what makes the responsibility assignment difficult. A decomposition algorithm is presented and a brief analysis is given.

In chapter 5 we give a preliminary treatment of learning by example. By borrowing results from chapter 4, we demonstrate difficulties in a few schemes available in the literature. An attempt to extend these strategies exposes the fundamental problem in any perceptron research, namely the extreme distributedness of the control and hence the necessity for global information to each node of the network.

Chapter 6 presents conclusions and directions for future research.

processing; no time sequences are involved. Boolean logic is therefore a simple but powerful formalism to represent parallel computations.

Perceptrons address at the simplest possible level (that of Boolean operations) the most crucial question in the development of parallel systems, namely *assignment of functional responsibility:* Decide what parts of a task must be assigned to which parts of a given system. More specifically, how does one accomplish a pattern classification task with a given multilayered network of adjustable Boolean nodes? This question has remained practically unanswered to this date. Our work on perceptrons, which directly addresses it, is therefore a contribution to functional responsibility assignment.

1.2. ADAPTATION AND LEARNING

The second key topic in the perceptron work is adaptation and learning. There are strong pragmatic reasons for studying learning in computational systems and for developing adaptive machines. The cost of programming computers is currently enormously large because little of the programming effort is automated. Learning techniques would alleviate this problem since a learning system infers the necessary steps in a computation from an abstract specification (for instance a set of examples) that does not include the internal computational steps. Moreover, it is an essential characteristic of intelligent behavior to be able to interact with an unknown or changing environment. Designing adaptive machines is therefore important in artificial intelligence research. In short, making computers more useful will

2.1. WHAT ARE PERCEPTRONS?

The term *perceptron* is not the name for one specific system, but it includes a broad class of neural network models. What is called a *simple perceptron,* sometimes referred to as the *classical perceptron,* is only an initial model that was studied to analyze and demonstrate some properties of the concept. Perceptrons in general, however, include a large class of other structures as well [Rose62a, Bloc62b, Rose62b]. Yet, many treatments of the subject consider the *simple perceptron* to be the only perceptron [Mins69, Rals83]. A clarifying discussion of this misinterpretation is given in [Bloc70]. In this dissertation we will use the term *perceptron* to denote the general class, as opposed to *simple perceptron* or *UCLA perceptron,* which refer to a restricted subset of perceptrons.

In this section, a formal definition of perceptrons is given first, followed by an explanation of its neuromimetic features and an overview of possible implementations.

2.1.1. Formal Definition

A perceptron is a multilayered network of polyfunctional combinational nodes. Each node i of the perceptron has k_i inputs, $\mathbf{x}_i=(x_{i,1}, \ldots, x_{i,k_i})$. The output z_i of node i is a Boolean function of the inputs:

$$z_i = f_i(\mathbf{x}_i) = f_i(x_{i,1}, \ldots, x_{i,k_i})$$

This function f_i of the node is adjustable (by some external means),

1.2.2. Perceptrons as Adaptive Devices

Adaptation implies a *multipurpose* system; a system cannot adapt if it implements only a single computation. Each node of a perceptron implements a Boolean function and adaptation consists of changing these functions. As a result of these changes, the pattern classification implemented by the perceptron is adjusted.

A perceptron learns and implements a Boolean function, which represents a classification of a number of simultaneously available inputs. The simplicity and implementation-independence of Boolean logic is also an advantage in describing learning. We refer to [Gall85b] and [Vali84] for a more detailed discussion of the representational power of respectively threshold logic and Boolean logic.

The classical perceptron work introduced a simple prototypical model of adaptation, namely learning by example. In our work, we suggest that the case where a complete specification of the task is given in one step needs to be understood first, instead of examining different possibilities for learning by example immediately. One-step learning is by far the simplest possible learning environment and the results of this study can be used later in other cases of learning.

The crucial issue in this context is again the question of *assignment of functional responsibility:* How can the local functions of the constituent nodes be adjusted such that the global network function matches the instructions, whether they be specified in one step or incrementally?

$\forall\ i,j$, either $\exists\ r$ such that $x_{i,j}=z_r$, or $\exists\ s$ such that $x_{i,j}=x_s$.

$\forall\ r$, either $\exists\ i,j$ such that $z_r=x_{i,j}$, or $z_r=z$.

x_s is a network input, $s \in \{1, . . . , n\}$; z is the network output. An example of a perceptron is shown in figure 2.2.

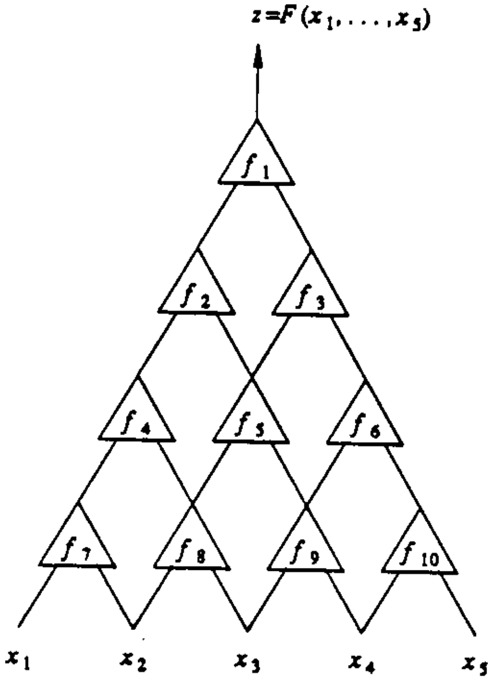


Figure 2.2: Example of a perceptron. It has 5 inputs and consists of 10 nodes with 2 inputs each.

In this dissertation, no loops are allowed in the perceptron network and it is assumed to have a single output. The resulting network then implements a Boolean function of n inputs $\mathbf{x}=(x_1, . . . ,x_n)$ and the network output z can be expressed as:

1.4. OVERVIEW OF THE DISSERTATION

This dissertation presents a unifying study of perceptrons, their applications, and the problem of functional responsibility assignment. An implementation-independent Boolean treatment is used throughout.

Chapter 2 gives a detailed overview of perceptrons. It begins with a formal implementation-independent definition of perceptrons: they are multilayered networks of fixed topology consisting of combinational nodes with adjustable logic. This definition incorporates the essential concepts, namely the functional flexibility of the nodes and the multilayeredness of the interconnections. Several implementations, both analog and digital, are then presented. Next, we review Rosenblatt's original perceptron research and show the extensiveness of his ideas. A particular class of digital perceptrons developed at UCLA is then discussed in detail. A review of other literature concludes the chapter.

Chapter 3 discusses a class of possible applications for perceptrons: combinational rule-based systems. More specifically, we show that, subject to some restrictions, a propositional-logic rule base can be transformed into a pair of Boolean functions. If speed and flexibility are important, such systems can be implemented with perceptrons. This chapter may be of interest by itself to some readers because it shows the relationship between rule-based and combinational systems.

Chapter 4 is a detailed treatment of one-step learning, namely the decomposition of a Boolean function such that it can be implemented with a

$$\gamma = \frac{Q}{2^{2^n}}$$

Since $0<Q \leq 2^{2^n}$ it follows that $0 < \gamma \leq 1$.

If the perceptron contains l nodes then $P = \prod_{i=1}^{l} p_i$ different combinations of node functions are possible. P is typically larger than Q and different combinations of node functions may result in the same global network function, that is, a given network function can be obtained by different combinations of nodal functions. This phenomenon is called *functional redundancy.* If $P > Q$ then the perceptron is said to be *redundant;* if $P = Q$ then it is *non-redundant.*

We define the *redundancy factor* ρ of a perceptron as follows:

$$\rho = \frac{P}{Q}$$

Since Q cannot exceed P it follows that $\rho \geq 1$.

2.1.2. Neuromimetics

An important motivation for the perceptron research is *neuromimetic* in nature. The object of neuromimetics is to develop computer systems or components that embody, to some practical extent, principles found in animal brains. A secondary goal of neuromimetics is to help in an understanding of the basic principles of the central nervous system.

Naturally, the internal operation of the brain is still largely unknown, but some general characteristics can be listed: fine-grain distributed

CHAPTER 2

OVERVIEW OF PERCEPTRONS

The principles behind the classical perceptron work have received extensive attention in the literature, up to the present time. Although many times the word *perceptron* is not used explicitly, the concept is fundamentally involved in most of the later work. Unfortunately, the same concepts that form the basis of so much work have also been misunderstood by many people. In this chapter we clarify the perceptron concept and expose the issues in their proper perspective.

Section 2.1 gives an introductory overview of perceptrons and shows the extensiveness of the idea. Section 2.2 presents a treatment of the early perceptron research. The well known *simple perceptron* is reviewed first, but some of the more general proposals are also presented. Section 2.3 is a more specific discussion of the perceptrons studied at UCLA. Finally, a short review of other perceptron-related research is presented in section 2.4.

therefore concerned with modeling simple reflexive interactions with the environment and not with complex ''thinking processes.''

2.1.3. Implementation Options

This section lists some options for implementing perceptrons. No specific perceptrons are reviewed (see sections 2.2, 2.3, and 2.4), but we present, in a general framework, alternatives for the nodes and the network interconnections.

2.1.3.1. The node

The node of a perceptron is a simplified model of the input-output behavior of a nerve cell (a neuron). Neurophysiology teaches us that in a neuron signals of different strengths are summed and cause an all-or-none response [Moun80]. The result is a unidirectional processing of input signals. The effect of incoming signals is subject to change over time (called adaptation, conditioning, habituation, etc.).

For practical and analytical purposes the model of a neuron must necessarily be *simplified,* yet it must be *good enough* to exhibit some of the unique properties of neural systems. In perceptrons the simplifying assumptions are twofold: the input and output signals are assumed to be entirely digital, and the neuron is assumed to be combinational. In other words, a nerve cell is considered to be a Boolean gate. The behavior of a neuron is modifiable, and so is the function implemented by the Boolean gate.

independent of the inputs and output of the node, and independent of other nodes. In other words, each node can implement any one of a set of possible functions:

$$f_i \in \phi_i = \{f_{i,1}, \ldots, f_{i,p_i}\}, \quad \text{where } |\phi_i| = p_i$$

If $p_i = 2^{2^{k_i}}$ then the node can implement *all* possible functions of its k_i inputs and it is called *complete* or *universal.* A schematic representation of the node is shown in figure 2.1.

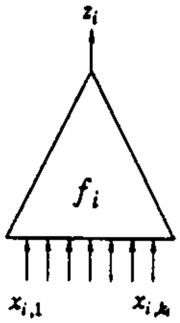
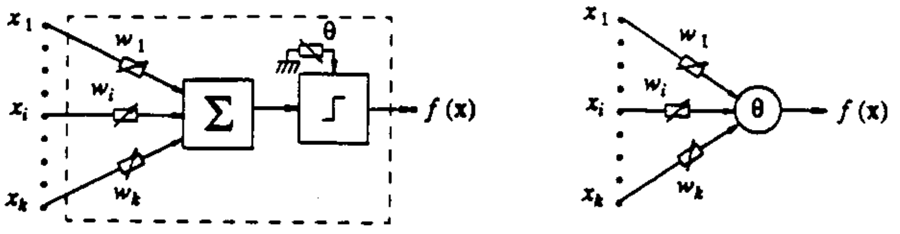


Figure 2.1: Schematic representation of a node (labeled *i*) of a perceptron.

The interconnections between the nodes are done in a hierarchical fashion, that is, the inputs of a node are connected to the outputs of other nodes or to the external environment. Similarly, the output of each node is connected to either the input of another node or to the external environment. In other words,



(a) (b)

Figure 2.3: Schematic representation of a threshold gate. (a) Detailed picture of the internal operations. (b) Simplified representation.

be done, for instance, by computing the output value for all 2^k possible input vectors of the threshold gate. However, the set of all threshold functions is *not* equal to the set of all Boolean functions of the same number of variables. For instance, the *exclusive-or* function

$$f(x_1,x_2) = x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$$

is not a threshold function [Vers82]. A threshold gate is therefore a functionally *incomplete* gate. With increasing number of inputs k the completeness factor of a threshold gate decreases and in the limit [Came60]:

$$\lim_{k \to \infty} \gamma < 2^{(k^2 - 2^k)}$$

An important characteristic of threshold gates is the analog nature of the functional adjustments, namely the values of $\mathbf{w}$ and θ. Historically, this model has received almost exclusive attention because of its close

$$z = F(\mathbf{x}) = F(x_1, \ldots, x_n)$$

where

$$F = f_1(f_2, f_3)$$

$$= f_1[f_2(f_4, f_5), f_3(f_6, f_7)]$$

$$= \cdots$$

A number of Boolean functions $\{f_i\}$ are composed to form a larger function F. By changing the functions of the constituent nodes, the network can implement a set of different Boolean functions:

$$F \in \Phi = \{F_1, \ldots, F_Q\}, \quad \text{where } |\Phi| = Q$$

Again, if $Q = 2^{2^n}$ then the network can implement *all* possible functions of its inputs and is said to be *complete* or *universal.*

A perceptron implements a Boolean function and hence it classifies its inputs into two classes:

$$C_0 = \{\mathbf{x} \mid F(\mathbf{x}) = 0\}$$

$$C_1 = \{\mathbf{x} \mid F(\mathbf{x}) = 1\}$$

where $F(\mathbf{x})$ is the Boolean function implemented by the perceptron. There is a direct relationship between Boolean logic and digital pattern classification; one is equivalent to the other.

We define the *completeness factor* γ of a perceptron as follows:

Boolean functions), but *how* the Boolean function is selected. In other words, the difference lies in the *control interface.*

To illustrate these similarities and differences consider a 2-input threshold function

$$z = f(x_1, x_2) = [w_1 x_1 + w_2 x_2 \geq \theta]$$

For different values of weights and threshold this function corresponds to different Boolean functions. Table 2.1 lists all 2-input Boolean functions and for each of them a possible threshold function (if it exists).

Boolean Function	Threshold Function
0	$x_1+x_2\geq 3$
$x_1 x_2$	$x_1+x_2\geq 2$
$x_1 \bar{x}_2$	$x_1-x_2\geq 1$
x_1	$x_1\geq 1$
$\bar{x}_1 x_2$	$-x_1+x_2\geq 1$
x_2	$x_2\geq 1$
$x_1 \oplus x_2$	*none*
x_1+x_2	$x_1+x_2\geq 1$
$\bar{x}_1 \bar{x}_2$	$-x_1-x_2\geq 0$
$x_1 \otimes x_2$	*none*
$\bar{x}_2$	$-x_2\geq 0$
$x_1+\bar{x}_2$	$x_1-x_2\geq 0$
$\bar{x}_1$	$-x_1\geq 0$
$\bar{x}_1+x_2$	$-x_1+x_2\geq 0$
$\bar{x}_1+\bar{x}_2$	$-x_1-x_2\geq -1$
1	$x_1+x_2\geq 0$

Table 2.1: Implementation of the 2-input Boolean functions as a linear threshold function.

processing and memory, a layered and hierarchical structure, largely regular interconnections, adaptiveness, etc. The specifics of these attributes are unclear, yet it is worthwhile to evaluate their usefulness for the design of computer systems.

Rosenblatt proposed perceptrons as a model of biological pattern recognition and partly as a proposal for a special-purpose computer [Rose58]. But in later publications he stresses that brain modeling is the major goal of his work [Rose62a].

More recent perceptron work, including ours, tends to diverge from an abstract brain modeling goal towards a more pragmatic objective. Rosenblatt must be credited, in his attempt to model the brain, for posing the right questions, discovering the important principles, and proposing a basic implementation. It is up to today's computer engineers to incorporate these concepts in practical systems.

One question addressed by Rosenblatt is *how* and *in what form* is information stored in the brain, and how does this information influence later behavior? It is generally accepted that experiences are not explicitly stored in a centralized memory, but that these experiences facilitate particular responses and that information is contained in the many connections in the stimulus-response chain. The cells in this chain can change their function and learn by making use of positive or negative feedback. In summary, the memory or state of a perceptron corresponds to the particular *function* it executes; its knowledge is *how to react* to stimuli. Perceptron research is

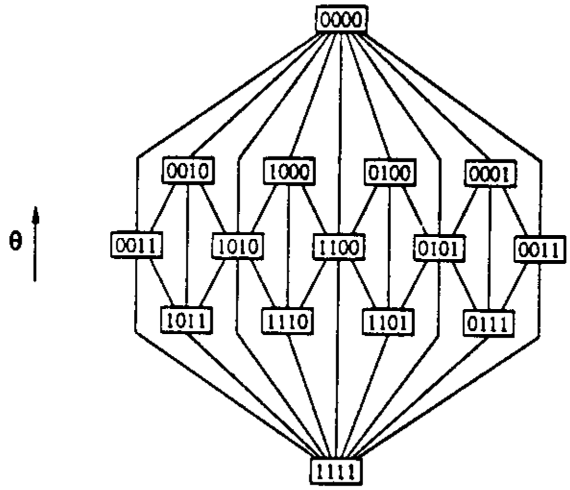


Figure 2.5: Functional transitions when changing the threshold of a 2-input threshold gate.

is also important for practical implications. The threshold gate model is derived from an analog implementation and a digital realization would be complex. The function-set model, on the other hand, lends itself to a digital implementation. Possible physical implementations of these Boolean models are now presented.

Figure 2.6.a shows a *Random Access Memory* (RAM) with k address lines and 2^k bits of memory. The 2^k bits of the RAM store the truth table of the Boolean function selected for that node, which is therefore *complete*. By presenting an input vector $\mathbf{x}$ on the address lines, the RAM reads one bit from memory. Changing the Boolean function of the node is done by rewriting the appropriate bits in the RAM. This requires selecting the bits to be changed via the address bus, setting the z-line to the desired value, and relationship to the sum and threshold operation of neural cells. The analog interface is also closely related to the adaptation in living systems, namely the process of reinforcing or depressing certain inputs by increasing or decreasing the associated weights, or the process of raising or lowering the activity level of a neuron by decreasing or increasing its threshold. However, from a pragmatic point of view, a threshold gate is a Boolean gate, and a fully digital model may be more appropriate for implementation.

b. Function-set model

In our work the node of a perceptron is a polyfunctional unit that can implement any one of a set of Boolean functions. A particular function of ϕ is *assigned* to a node by a process called *control*. In the function-set model, the control occurs in an abstract way, independent of any particular implementation.

This model is more flexible in that it allows different choices for the set of nodal functions ϕ. In a threshold gate model ϕ is always the set of threshold functions, but in a function-set model ϕ could be any set. For example, it could be complete.

Although this model includes the threshold gate model (indeed, any threshold function is a Boolean function), the conceptual difference is important. Working with weights and threshold is different from selecting a Boolean function from a set. The difference between the threshold gate model and our model is not *what* is accomplished (both implement a set of table can be done without interfering with the data processing. The *control process* is totally separate from the *data processing*. The separation of control and data is implied by the perceptron definition and is also present in the threshold gate model. The ULM-based implementation is therefore more appropriate. On the negative side, a ULM implementation requires more control lines than a RAM.

A Boolean node with an incomplete set of functions can also be implemented with digital circuitry. For example, Armstrong introduces 2-input nodes that implement only the *nonconstant increasing* functions (see table 2.2) [Arms78]. An implementation of this node is given in figure 2.7.

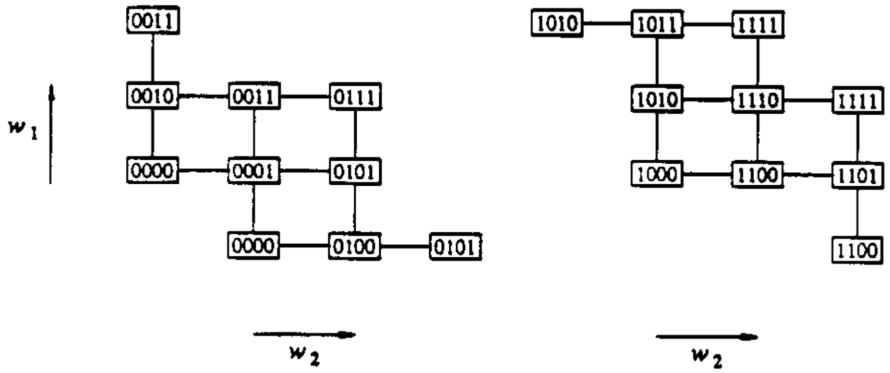
c_1	c_2	$f(x_1,x_2)$
0	0	x_1x_2
0	1	x_1
1	0	x_2
1	1	x_1+x_2

Table 2.2: Selection of one of the four 2-input nonconstant increasing functions.

2.1.3.2. The network

A perceptron is a model of a neural network. A number of nodes are interconnected in a layered fashion, corresponding to the present-day understanding of neural systems. The *simple perceptron*, for instance, is a two-layer network of threshold gates.

If only the weights of the threshold gate are changed then the corresponding Boolean function obeys the transitions shown in figure 2.4.a and 2.4.b. In this figure each function is represented by a string of 4 binary digits "$z_0 z_1 z_2 z_3$," where $z_0 = f(0,0)$, $z_1 = f(0,1)$, $z_2 = f(1,0)$, $z_3 = f(1,1)$. Moving upwards in this transition graph corresponds to an increase in w_1. The same holds for w_2 in the horizontal direction. The set of 14 threshold functions is separated into 2 disconnected subsets by a change in sign of the threshold value.



(a) θ>0 (b) θ<0

Figure 2.4: Functional transitions when changing the weights of a 2-input threshold gate.

The transitions when modifying the threshold are shown in figure 2.5. This diagram shows how the transitions from *always true* (1111) to *always false* (0000) can be achieved in 1, 2, or 3 steps.

The difference between a threshold gate model and a function-set model

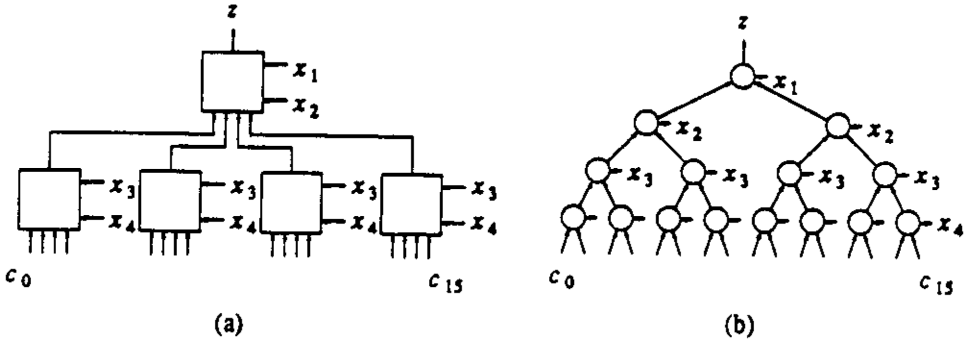


Figure 2.8: Typical example of a ULM tree (a) and a binary decision tree (b).

from the process of *selecting* this function. Both modes can be envisioned as being *orthogonal* to each other (see figure 2.9). Only when F must be changed gradually (as a result of adaptation or learning) do both processes interact. In a ULM tree or a binary decision tree, no separation of control and data is present.

There are several options for the interconnection topology of a perceptron. Some of the earlier perceptron research dealt with a largely random topology, partly because this perceptron was only a simple initial prototype. The reason for the randomness was also partly because, at that time, it was believed that the human brain was indeed interconnected in a largely random way. Later research examined more structured topologies using specific interconnections for detecting certain features (lines, corners, etc.). The advantage is improved efficiency. The disadvantage is their special-purpose nature and therefore restricted application domain.

toggling the READ/WRITE port. The RAM was used in the work of Aleksander [Alek79].

Figure 2.6.b shows a slightly different implementation. It shows a 2-input *Universal Logic Module* (ULM) with attached to it a 4-bit register. A ULM is a multiplexer: presenting an input pattern to the x-lines selects one of the 4 bits of the register, and the selected value is routed to the z-line. The function is changed by modifying the contents of the 4-bit control register via the separate c-lines. A k-input node requires a $k \times 2^k$ ULM and a 2^k-bit register. We refer to [Yau68, Yau70] for more details about ULMs.

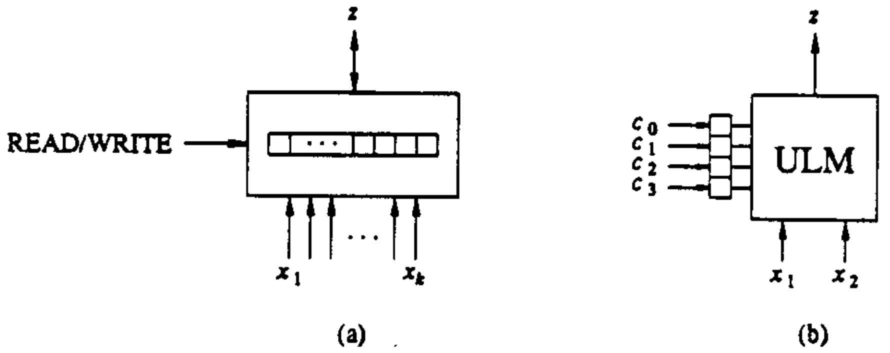


Figure 2.6: Two digital nodes of a perceptron. (a) RAM. (b) ULM.

A ULM node is similar in operation to a RAM, except that in the former the data processing and function selection are totally independent of each other. In a RAM, the Boolean function is changed by using the input and output lines, and normal data processing must be halted while the function is being modified. In a ULM node, on the other hand, the changes in the truth

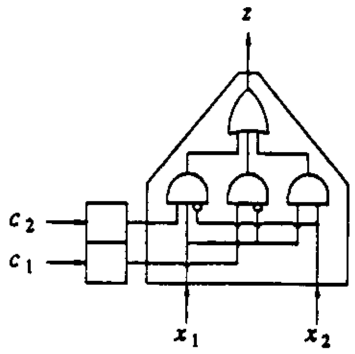


Figure 2.7: Example of a digital node that implements only the 2-input nonconstant increasing functions.

With a digital node, however, the class of perceptron networks is sometimes misunderstood. In a ULM node, the inputs and output are digital, but so are the four control lines that select the function. It is therefore possible to interconnect the output of one node to the control lines of another. A typical example of such an approach is the work on ULM trees (figure 2.8.a) [Yau68, Yau70, Ston71] or binary decision trees (figure 2.8.b) [Aker78, Cern79, Mato83]. This class of networks is not included in the definition of perceptrons.

A similar approach is not possible with threshold gates since the control of a threshold gate (the weights and threshold) is analog and can therefore not be connected to the digital outputs of other nodes.

The definition of perceptrons introduces the important concept of a clear and explicit separation of *data* and *control*. The data processing mode of a perceptron, that is, the computation of $z=F(x_1, \ldots, x_n)$, is entirely separate

2.2.1. The Simple Perceptron

2.2.1.1. Structure

The structure of the *simple perceptron* is shown in figure 2.11. On the left is a set of transducers for physical signals outside the network. It would typically be an array of light sensors (a retina). When a sensor is excited it produces an output signal 1, else it signals a 0. The outputs of the sensors are connected to a layer of threshold gates called *association units* or *A-units*. These connections are generated randomly. The weights and threshold of all the A-units are selected a priori (for instance at random) and are never changed once the system is built. The outputs of the A-units are connected to a single threshold gate, called the *response unit* or *R-unit*, which has variable weights and a variable threshold. Functional flexibility of the *simple perceptron* is therefore restricted to this node.

Constraining the flexibility of the system to the R-unit is a strong restriction, but it was imposed for reasons of simplicity. The theoretical analysis of a *simple perceptron* reduces to the study of the capabilities and adjustment procedures of a single threshold gate.

The association layer implements a *preprocessing* of the raw input data and generates *features* of these inputs. If the interconnections are random then the features produced by the A-units are equally random. By not imposing any specific interconnections, and hence features, it is hoped that the necessary feature detectors for the given application are present. In other

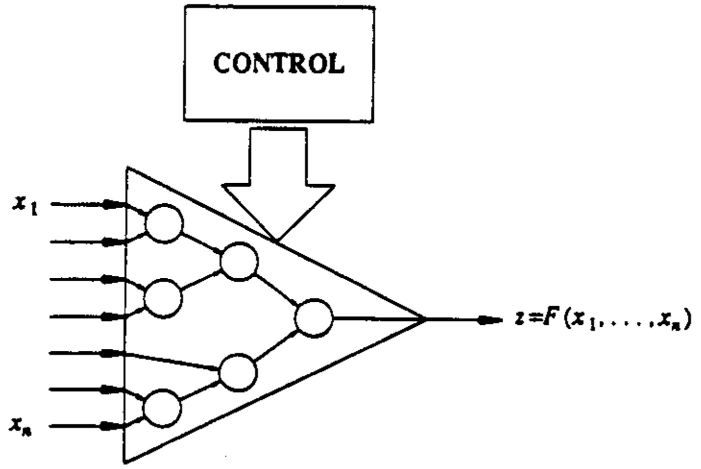


Figure 2.9: Separation of data processing and function selection in perceptrons.

The interconnection topology favored in our research at UCLA is one of maximum *regularity*. This means that the network is generated by a systematic replication of a basic interconnection principle [Malo82]. A typical example of a UCLA perceptron is the triangular network shown in figure 2.10. A regular interconnection topology has many advantages for a VLSI implementation [Moor85a].

2.2. THE CLASSICAL PERCEPTRON RESEARCH

The earliest perceptron work (1957-1964) was conducted by Rosenblatt and his group at Cornell Aeronautical Laboratory in Buffalo, NY, and Cornell University in Ithaca, NY. It was an extensive research effort with many good ideas, unfortunately many unfinished.

layer and the association layer were possible.

2.2.1.2. Learning

Although learning in perceptrons is not discussed until chapter 5, a treatment of the *simple perceptron* would be incomplete if it did not briefly present its capabilities for learning by example.

The *simple perceptron* learns in the following way. Initially the values of the weights and the threshold of the R-unit are arbitrary. Example inputs are presented by a *teacher* and the output of the R-unit is compared with the desired value. If the perceptron produces the right output then nothing happens. However, if an incorrect output occurs then the teacher sends a negative feedback to the perceptron. A *reinforcement rule* then increases or decreases the weights or threshold of the R-unit. When enough examples have been shown, a set of weights and threshold results that correctly classifies the examples, provided such a set exists.

An important premise in the algorithm is the assumption that an appropriate set of weights and threshold exists. This means that, obviously, the learning scheme cannot find a solution if none exist. The work addresses *how* an implementation of a task (a Boolean function) can be found by trial and error. The question of *which* Boolean functions a given *simple perceptron* can achieve is a different issue, and is reviewed next.

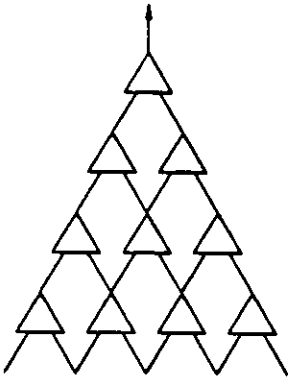


Figure 2.10: A perceptron with a regular interconnection geometry.

In the more recent literature, however, there is some misunderstanding about the class of perceptrons introduced in the early work. Rosenblatt pointed out that what is sometimes referred to as *the* perceptron is only a *simple perceptron,* a small prototype of a large class of neuromimetic systems. He explicitly objected to writing the word *perceptron* with a capital letter and stressed that the term *perceptron* was "... intended as a generic name for a variety of theoretical nerve nets" [Rose62a].

The section below discusses the *simple perceptron* and its limitations. Many of these limitations, however, do not apply to other, more powerful, perceptrons, which are discussed in the second section.

The main weakness of the *simple perceptron* is a lack of computational power. A possible improvement consists of adding more layers to the network, that is, using more depth and less width. Additionally, more than one node or layer of the perceptron should be adjustable.

2.2.1.4. Summary

The *simple perceptron* is a prototype perceptron that shows the feasibility of a simple strategy for learning by example. It is, however, too restricted and too inefficient for many advanced tasks. The characteristics of the *simple perceptron* that contribute to its deficiencies can be summarized as follows:

1. It has only two layers of functional nodes.
2. The interconnections are largely random.
3. The nodes have a large number of inputs, which grows with the number of network inputs.
4. The nodes can only implement threshold functions, which is an incomplete set.
5. Only one node, the R-unit, is adjustable and contributes to the flexibility of the perceptron.
6. The nodes have analog internals and are difficult to implement with digital circuitry.

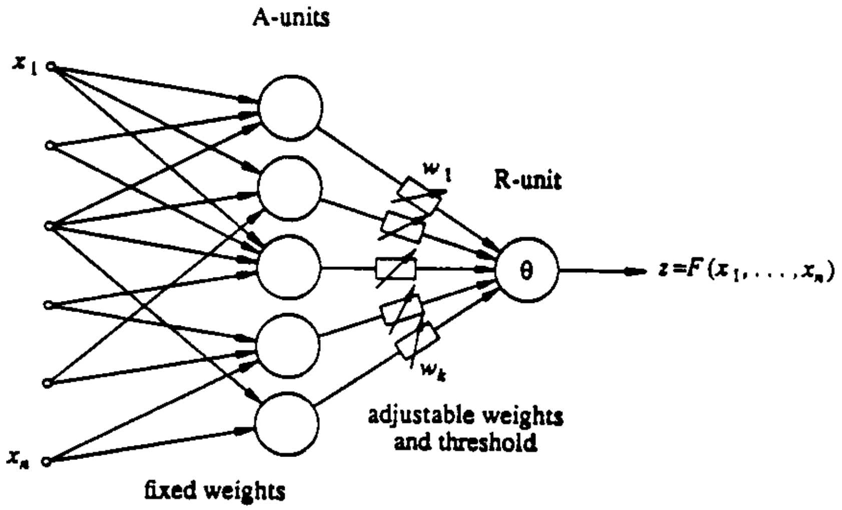


Figure 2.11: Structure of the *simple perceptron.*

words, randomness is chosen in the hope of maximizing the completeness factor of the perceptrons. However, this approach is inefficient and more structured interconnections improve the capabilities of perceptrons [Rose59, Rose62b].

Many experiments examined the performance of this *simple perceptron.* Simulation experiments are described in [Rose60a]. More importantly, a prototype, called the *MARK I perceptron,* was built at the Cornell Aeronautical Laboratory and used in experiments [Hay60]. The system used electromechanical integrators and transistor-driven relay circuits. Its input layer consisted of a 20×20 square of photosensitive cells connected to a camera. Alternatively, 400 toggle switches could be used to assign values to its inputs directly. There were 512 A-units. 16,000 connections between the sensory

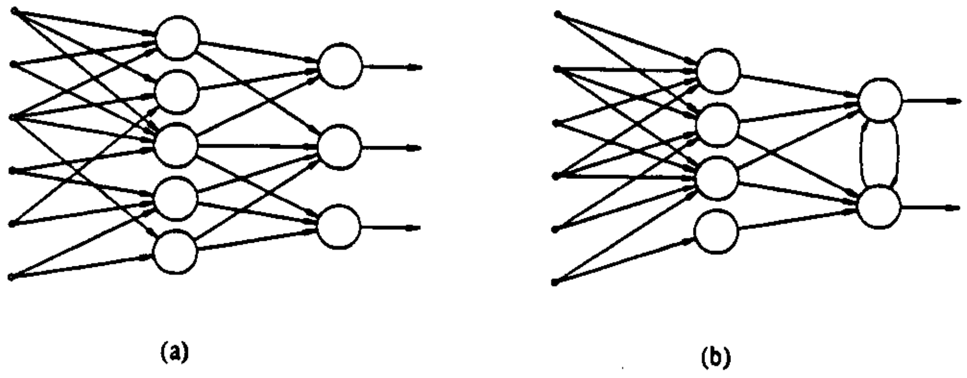


Figure 2.12: Example of a multi-output (a) and a dual-output (b) perceptron.

2.2.2.2. Cross-coupled and back-coupled perceptrons

Cross-coupling means interconnecting different nodes of the same layer (figure 2.13.a). Such perceptrons were proposed in [Rose58, Rose60b, Bloc61]. *Back-coupling* means connecting the output of a node to an input of one or more nodes in layers closer to the inputs, for instance connecting the output of the R-unit to an input of an A-unit (figure 2.13.b). Examples can be found in [Rose64]. Such structures are excluded from our treatment here.

2.2.2.3. Multilayer perceptrons

Perceptrons with multiple layers of adjustable nodes were reported in [Rose60b, Bloc61, Bloc62b, Konh62, Rose62b]. Such perceptrons are difficult to treat formally, and no satisfactory scheme for learning by

2.2.1.3. Limitations

Minsky and Papert [Mins69], and others [Uesa75, Abel77], examined the functional limitations of the *simple perceptron*. Specifically, they showed what geometrical patterns can or cannot be classified by the *simple perceptron*.

Minsky and Papert's work is a study of the incompleteness of a *single* threshold gate, namely the R-unit of a *simple perceptron*. Since the R-unit is restricted in its functional capabilities, certain parts of the function intended for the perceptron must be assigned to the gates in the association layer. Minsky and Papert examined how much of the perceptron function can be implemented by the R-unit, and interpreted these results in geometrical terms. In other words, they did a geometrical interpretation of the incompleteness of the threshold gate. Their work formally proved that many of the random connections between input sensors and A-units (as allowed by Rosenblatt) produce a perceptron that is unable to classify some geometrically simple input patterns.

The *simple perceptron* has a fixed preprocessing layer and one decision element that makes its decision according to a learned linear rule. To achieve a larger functional capability, the perceptron must have more A-units, and hence the R-unit must have more inputs. Yet, the completeness of a threshold gate decreases rapidly with increasing number of inputs. As a result, the functional capabilities of a *simple perceptron* are severly constrained.

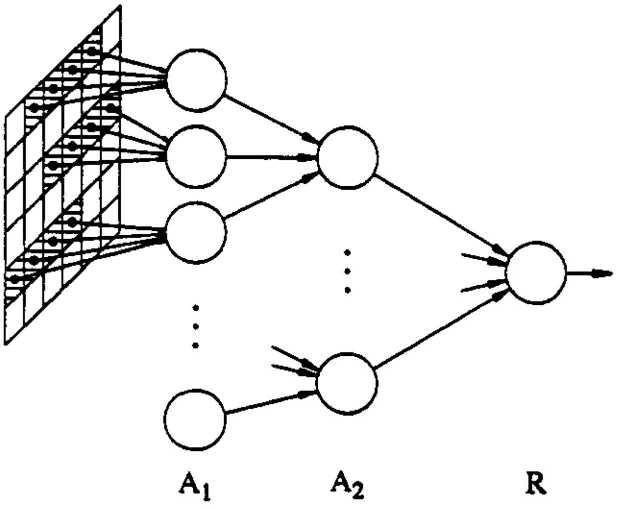


Figure 2.14: Example of a structured multilayered perceptron.

2.2.2.4. Perceptrons with different nodes

Linear threshold gates are not the only possible building blocks for perceptrons [Rose62a]. Rosenblatt mentioned for example *continuous units* that have no threshold operation and therefore have analog outputs. More interestingly, he also suggested to replace the weighted sum of a threshold gate by a more general analog function of weights and input values. These nodes are called ***nonlinear threshold gates***, as opposed to *linear threshold gates*.

EXAMPLE 2.1: A 2-input linear threshold function

$$f(x_1,x_2) = [w_1x_1+w_2x_2 \geq \theta]$$

can represent only 14 of the 16 2-input Boolean functions. On the other

2.2.2. Other Perceptrons

Rosenblatt suggested possible ways of improving the functional characteristics of the *simple perceptron* and he developed many extensions. Some of these more general perceptrons are reviewed in [Rose62a, Rose62b]. The overly negative impression that currently surrounds the perceptron work is a result of an ignorance of these more general systems and a too concentrated focus on the limitations of the simplest initial prototype of a large class. See for example [Bloc70] for a discussion of this misunderstanding.

A brief overview of these more general perceptrons is given below.

2.2.2.1. Multi-output perceptrons

An obvious extension of the *simple perceptron* consists of connecting more than one R-unit to the same association layer (figure 2.12.a). Such perceptrons were already included in the earliest perceptron literature [Rose58]. For example, the MARK I perceptron had 8 R-units [Hay60].

If only the R-units of such a perceptron are adjustable then the same straightforward learning algorithm as in the *simple perceptron* applies. However, if the nodes of the association layer participate in the functional change as well then a more general theory is necessary. Multi-output perceptrons are not treated in our study, and we also exclude *dual-output* perceptrons. In the latter structures, two R-units inhibit each other such that only one can output a 1 (figure 2.12.b). This is an implementation of the well known *winner-take-all* concept.

2.3. THE UCLA PERCEPTRONS

The goal of our research at UCLA is to develop a general but practical model of perceptrons and to investigate the functional properties and learning capabilities of a few small prototypes. The learning aspects are the subject of chapter 5; in this section we concentrate on the functional capabilities.

The reader is referred to [Vers82, Vers83] for some precursor work, and to [Moor83, Sala83, Moor85a, Moor85b, Moor85c, Moor85d] for some practical implementation issues. [Vida83, Vida85] outline the general scope and goals of the research.

2.3.1. Structure

2.3.1.1. The node

The node of the UCLA perceptrons is a direct implementation of a complete function-set model. Each node consists of a 2-input Universal Logic Module (ULM), as described in section 2.1.3.1. A ULM implements the following Boolean function:

$$f = c_0\bar{x}_1\bar{x}_2 + c_1\bar{x}_1x_2 + c_2x_1\bar{x}_2 + c_3x_1x_2$$

It is a multiplexer selecting one of the c-lines as specified by the x-lines (figure 2.15.a).

In our perceptron design the ULM is used sideways. The *control lines* (c-lines) represent the entries of a truth table and specify a Boolean function of the *data lines* (x-lines), which now become the real inputs (figure 2.15.b).

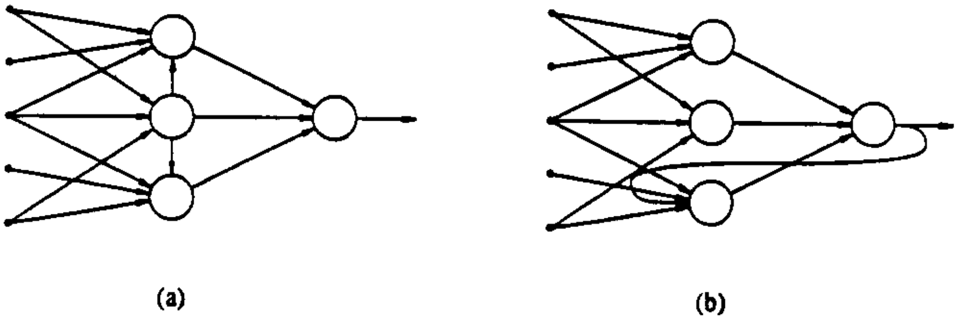


Figure 2.13: Example of cross-coupling (a) and back-coupling (b) in perceptrons.

example exists. Therefore, the connections are designed to implement specific feature detectors necessary for a particular application. For example, the first layer A_1 might extract lines and the next layer A_2 might combine these outputs into position-independent features (figure 2.14). Such a hierarchy of *property detectors* allows a more focussed data processing and a more directed learning. This approach was later supported by results obtained by Hubel and Wiesel in their neurophysiological work [Hube62].

In summary, more general perceptrons are more powerful, but require more complex learning schemes. No general theory exists and hence special-purpose systems were developed.

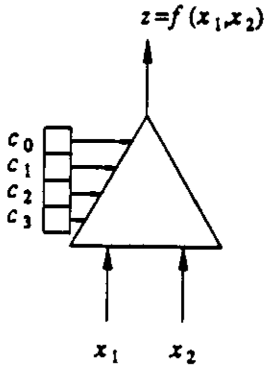


Figure 2.16: The basic node of a UCLA perceptron.

also redundant.

Figure 2.17.b shows an assignment to the nodes such that the network implements the following Boolean function:

$$z = F(x_1,x_2,x_3) = x_1x_2\bar{x}_3+\bar{x}_1x_2x_3$$

In this figure (and in others that will follow) we use the notation of table 2.4 to specify the functional assignments.

The UCLA perceptrons include a large class of systems and not just one prototype. An example of a 4-input perceptron is shown in figure 2.18. This perceptron, however, is incomplete.

hand, a polynomial threshold function of the form

$$f(x_1,x_2) = [w_1x_1+w_2x_2+w_{12}x_1x_2\geq\theta]$$

can generate all 16 functions (see table 2.3). □

The set of polynomial threshold functions constitutes a much richer set than linear threshold functions. Polynomial threshold functions of any number of inputs can form a complete set if the degree of the polynomial is large enough.

Truth table				Boolean Function	Linear Threshold Function	Polynomial Threshold Function
0	0	0	0	0	$x_1+x_2\geq3$	$x_1+x_2\geq3$
0	0	0	1	x_1x_2	$x_1+x_2\geq2$	x_1+x_2
0	0	1	0	$x_1\bar{x}_2$	$x_1-x_2\geq1$	$x_1-x_2\geq1$
0	0	1	1	x_1	$x_1\geq1$	$x_1\geq1$
0	1	0	0	$\bar{x}_1x_2$	$-x_1+x_2\geq1$	$-x_1+x_2\geq1$
0	1	0	1	x_2	$x_2\geq1$	$x_2\geq1$
0	1	1	0	$x_1\oplus x_2$	*none*	$x_1+x_2-2x_1x_2\geq1$
0	1	1	1	x_1+x_2	$x_1+x_2\geq1$	$x_1+x_2\geq1$
1	0	0	0	$\bar{x}_1\bar{x}_2$	$-x_1-x_2\geq0$	$-x_1-x_2\geq0$
1	0	0	1	$x_1\otimes x_2$	*none*	$-x_1-x_2+2x_1x_2\geq0$
1	0	1	0	$\bar{x}_2$	$-x_2\geq0$	$-x_2\geq0$
1	0	1	1	$x_1+\bar{x}_2$	$x_1-x_2\geq0$	$x_1-x_2\geq0$
1	1	0	0	$\bar{x}_1$	$-x_1\geq0$	$-x_1\geq0$
1	1	0	1	$\bar{x}_1+x_2$	$-x_1+x_2\geq0$	$-x_1+x_2\geq0$
1	1	1	0	$\bar{x}_1+\bar{x}_2$	$-x_1-x_2\geq-1$	$-x_1-x_2\geq-1$
1	1	1	1	1	$x_1+x_2\geq0$	$x_1+x_2\geq0$

Table 2.3: Implementation of the 2-input Boolean functions as a linear or polynomial threshold function.

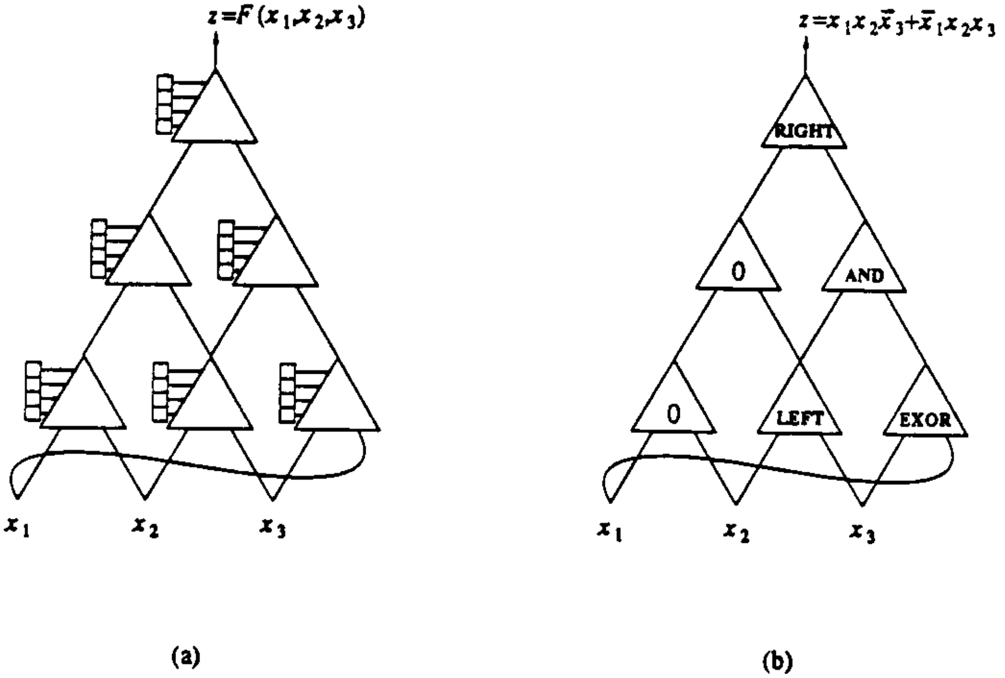


Figure 2.17: Typical 3-input UCLA perceptrons. (a) Structure. (b) Sample assignment.

2.3.2.1. Completeness

Completeness (sometimes referred to as *universality*) is a property of the functional set Φ of a perceptron. A *complete set* of Boolean functions of n inputs is a set that contains all 2^{2^n} Boolean functions of n inputs. A *complete perceptron* is one that can implement a complete set of functions.

The question of how to build a complete perceptron is a complex one and no systematic solution exists. Completeness in perceptrons has been studied for some specific threshold gate networks. For example, [Came60] shows a complete 3-input threshold gate network consisting of 3 nodes. A

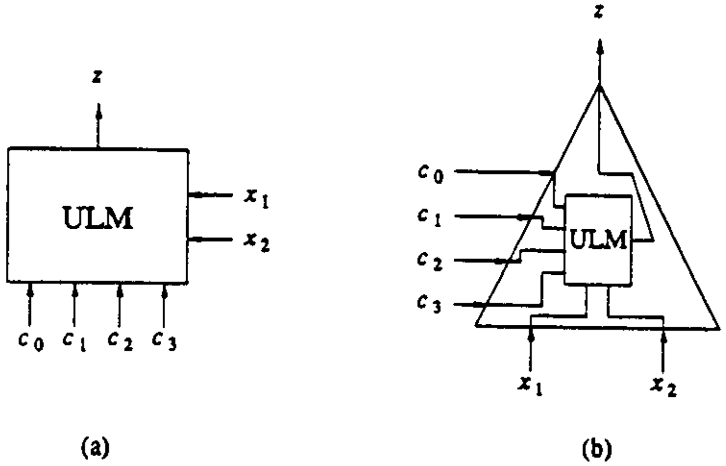


Figure 2.15: The ULM as the basic building block of the UCLA perceptrons. (a) Basic operation of the ULM as a multiplexer. (b) The ULM used as a functional node.

Additionally, the c-lines of a ULM are connected to a 4-bit local register that holds the desired truth table (figure 2.16). Such a node is functionally equivalent to a small RAM. It can implement any of the 16 2-input Boolean functions, listed in table 2.4.

2.3.1.2. The network

A typical example of a UCLA perceptron is shown in figure 2.17.a. The network has 3 inputs and consists of 6 nodes, each of which can implement 16 functions. This network is complete: it can implement 2^{2^3}=256 Boolean functions of 3 inputs. Because of this property it has sometimes been called a *General Purpose Perceptron* (GPP) in the past [Vers82]. The network is

$$P \geq Q$$

$$2^{4 \times l} \geq 2^{2^n}$$

$$4l \geq 2^n$$

$$l \geq 2^{n-2}$$

The number of nodes in a complete UCLA perceptron is always larger than 2^{n-2}.

An extensive set of 3-input perceptrons has been investigated for completeness [Vers82]. Figure 2.17 shows a complete perceptron. A proof of the completeness of this perceptron will be given in chapter 4. Another complete 3-input network is shown in figure 2.19.

Complete perceptrons could in theory be designed for any number of inputs (see for example figure 2.20), although it is not practical if n is large. The number of nodes involved in a complete network, as well as the number of layers, grows exponentially with the number of inputs. Therefore, if n is large, any perceptron is always incomplete.

This raises the question of the *incompleteness* of a specific perceptron. What is the functional set Φ of a given perceptron and how large is it? Such questions are difficult to answer in their full generality. A preliminary treatment can be found in [Urba68, Alek78, Vers82].

An example of an incomplete perceptron is the tree network shown in figure 2.21. This perceptron cannot implement for instance the following function:

c_0	c_1	c_2	c_3	$f(x_1,x_2)$	Name
0	0	0	0	0	0
0	0	0	1	x_1x_2	AND
0	0	1	0	$x_1\bar{x}_2$	AND NOT
0	0	1	1	x_1	LEFT
0	1	0	0	$\bar{x}_1x_2$	NOT AND
0	1	0	1	x_2	RIGHT
0	1	1	0	$x_1 \oplus x_2$	EXOR
0	1	1	1	x_1+x_2	OR
1	0	0	0	$\bar{x}_1\bar{x}_2$	NOR
1	0	0	1	$x_1 \otimes x_2$	EQUI
1	0	1	0	$\bar{x}_2$	NOT RIGHT
1	0	1	1	$x_1+\bar{x}_2$	OR NOT
1	1	0	0	$\bar{x}_1$	NOT LEFT
1	1	0	1	$\bar{x}_1+x_2$	NOT OR
1	1	1	0	$\bar{x}_1+\bar{x}_2$	NAND
1	1	1	1	1	1

Table 2.4: The 16 possible Boolean functions of 2 inputs.

2.3.2. Functional Characteristics

Two functional characteristics are important in the context of perceptrons: *completeness* and *redundancy*. Some general notions of these issues were presented earlier; this section gives more specific details and examples.

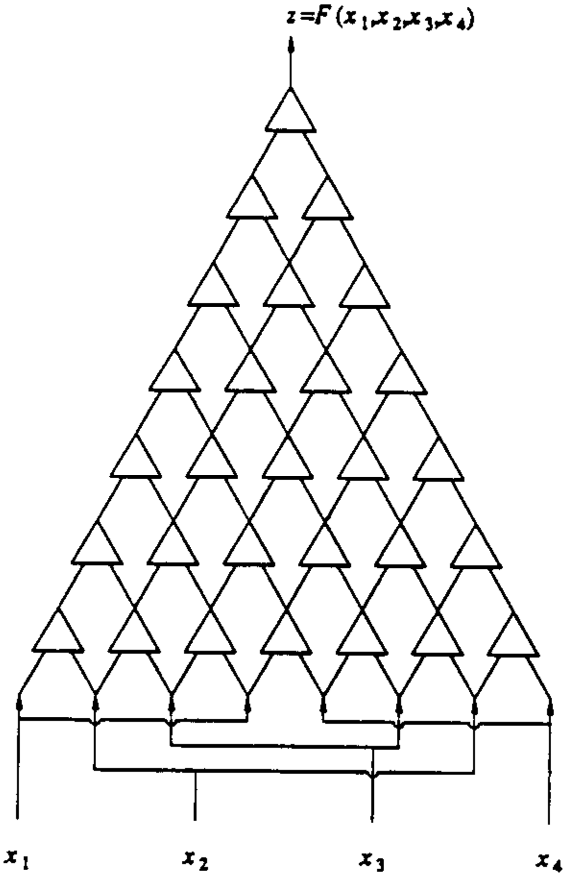


Figure 2.20: A complete 4-input perceptron.

$$z = F(x_1,x_2,x_3) = x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3$$

can be achieved many different ways, for example

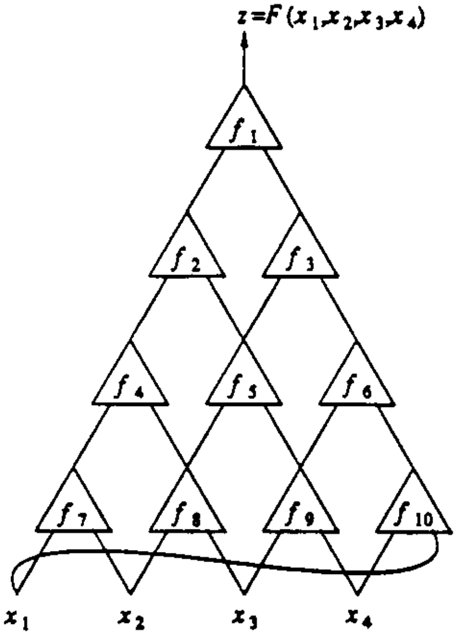


Figure 2.18: An example of a 4-input UCLA perceptron.

ULM tree (section 2.1.3.2) is an example of a complete logic network, but it is not a perceptron.

In UCLA perceptrons, a minimum requirement for completeness is that the total number of control bits (4 per node) be larger than 2^n. Indeed, the number of different internal states of the network is $P=2^{4\times l}$ (l is the number of nodes). If the perceptron is complete then $Q=2^{2^n}$ and since P is an upper limit for Q:

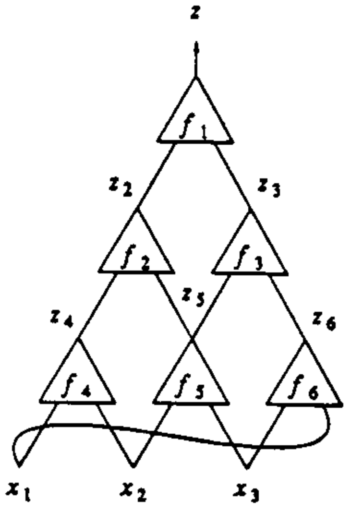


Figure 2.22: Complete 3-input perceptron.

$$\rho = \frac{P}{Q} = \frac{16^6}{2^{2^3}} = \frac{2^{24}}{2^8} = 2^{16}$$

Table 2.5 gives a partial list of the number of different implementations for the Boolean functions of 3 inputs, as determined by experiment [Moor85c].

Redundancy is important because it implies a potential for fault recovery: certain functions can still be implemented on the perceptron if a node malfunctions. Let us assume that, when a node malfunctions, its output is stuck at some constant value and that its parent node must ignore this input. Through experimentation we have found that a malfunction in any of the nodes of figure 2.22, except the top node, always reduced Φ to 192 function. This means that any malfunction removes only 64 functions from the set Φ, unless the top node fails.

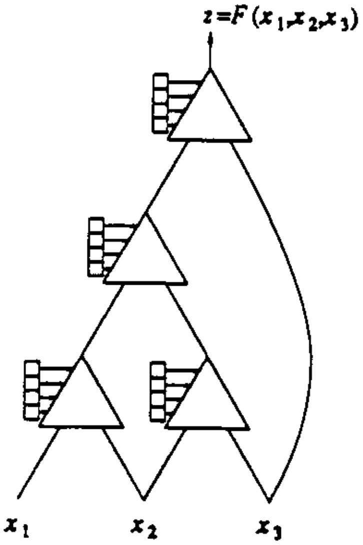


Figure 2.19: Alternative complete 3-input UCLA perceptron.

$$z = F(x_1,x_2,x_3,x_4) = x_1x_2+x_2x_3+x_3x_4$$

Chapter 4 will show why this is so.

2.3.2.2. Redundancy

Redundancy is another important topic in the study of perceptrons. One Boolean function may have more than one implementation on the same network, or vice versa, changing the internal assignments of a network does not necessarily change the network function. In perceptrons redundancy is the rule rather than the exception.

EXAMPLE 2.2: Consider the perceptron of figure 2.22. The network function

Which functions are affected by a particular fault, and *how many* faults can each function tolerate? The answer to this question is summarized in table 2.6, which groups the 256 functions according to what malfunction they can tolerate. The first five columns list the nodes with a single fault; the last column summarizes how many functions can tolerate a malfunction in these nodes. For example, the first row shows that 24 functions cannot tolerate any fault in the network. 8 other functions can only tolerate a fault in node 5 and not in any other node (second row). 8 functions can tolerate a single fault in either node 2 or node 4; 8 others can tolerate a fault in node 3 or 6; and so on. 136 functions can tolerate a single fault anywhere in the network, except the top node.

The 24 functions that cannot tolerate a single fault can be grouped into two categories represented by the functions:

$$f(x_1,x_2,x_3)=x_1x_2+x_2x_3+x_3x_1$$

$$f(x_1,x_2,x_3)=x_1x_2\oplus x_2x_3\oplus x_3x_1$$

In the context of fault recovery, two types of redundancy must be distinguished.

a. Trivial redundancy.

Consider an interconnection of the output of one node of a perceptron to the input of another node (figure 2.23). The following operation will not change the network function: negate the output of the first node (change f_1 to $\bar{f}_1$) and invert the corresponding input of the second node (change

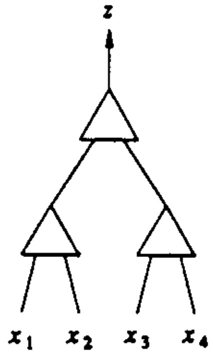


Figure 2.21: A binary tree perceptron with 4 inputs.

$f_1=z_2+z_3; f_2=z_4z_5; f_3=\bar{z}_5z_6; f_4=x_1x_2; f_5=x_2\oplus x_3; f_6=x_3\bar{x}_1$.

$f_1=z_2\bar{z}_3; f_2=z_4\oplus z_5; f_3=z_6; f_4=x_1x_2; f_5=x_2x_3; f_6=x_3x_1$.

$f_1=z_2+z_3; f_2=z_4\oplus z_5; f_3=z_5\bar{z}_6; f_4=x_1x_2; f_5=x_2x_3; f_6=x_3x_1$.

$f_1=z_3; f_2=0; f_3=z_5z_6; f_4=0; f_5=x_2; f_6=x_3\oplus x_1$.

etc. □

In figure 2.22, each of the structure's 6 atoms can implement 16 Boolean functions, hence $P=16^6$. This perceptron implements $Q=256$ different functions. The redundancy factor is

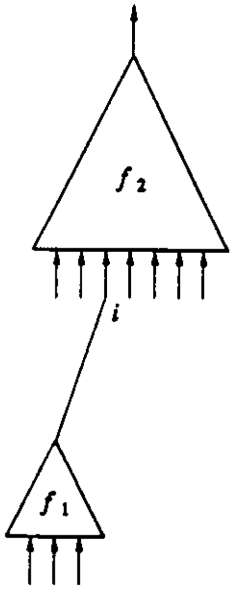


Figure 2.23: Example of trivial redundancy in a perceptron.

b. Nontrivial redundancy.

A more important form of redundancy is shown in figure 2.24. This figure shows how one function, namely $F(x_1,x_2,x_3)=x_1x_2\bar{x}_3+\bar{x}_1x_2x_3$, can be implemented in two entirely different ways. One implementation cannot be derived from the other by a trivial operation such as negating inputs or outputs. Other implementations were listed in example 2.2.

The cause of this redundancy is the fan-out at the inputs and inside the network. Nontrivial redundancy is a much more powerful property for fault recovery than trivial redundancy.

Function number	Function truth table	Number of implementations
0	00000000	3261376
1	00000001	128192
2	00000010	128192
3	00000011	113600
4	00000100	128192
5	00000101	113600
6	00000110	19776
7	00000111	22720
8	00001000	128192
9	00001001	19776
10	00001010	113600
11	00001011	22720
12	00001100	113600
13	00001101	22720
14	00001110	22720
15	00001111	127168
16	00010000	128192
17	00010001	188608
18	00010010	22528
19	00010011	20608
20	00010100	22528
21	00010101	20608
22	00010110	2496
23	00010111	2624
.	.	.
.	.	.
.	.	.
250	11111010	113600
251	11111011	128192
252	11111100	113600
253	11111101	128192
254	11111110	128192
255	11111111	3261376

Table 2.5: The number of different implementations on the complete 3-input UCLA perceptron of figure 2.22 for all 3-input Boolean functions.

is much simpler to deal with, as will become clear in chapter 4.

2.4. OTHER PERCEPTRON RESEARCH

2.4.1. Neuron Models

2.4.1.1. Threshold gate model

The history of the threshold gate as a simplified model of the neuron goes back to the seminal paper of McCulloch and Pitts [McCu43]. Based on the apparently *all-or-none* activity of a neuron, it was postulated that neurons communicate through binary signals. The behavior of a neural net is then expressible in Boolean algebra. The internal operation of a nerve cell was assumed to be one of excitation and inhibition (weighted sum) combined with a threshold operation. Hence the threshold gate as a neural model. McCulloch and Pitts used both open-loop and closed-loop systems. They did not, however, take advantage of the modifiability of the threshold gate function. In summary, their work introduced two simplifications, namely the use of Boolean algebra in the study of neural models, and the threshold gate as a model of the neuron itself.

Rosenblatt was the first to exploit these ideas and to propose a working model of a pattern recognizing neural network (later built in hardware). He was also the first to introduce the concept of a *changing* neural net by modifying the weights or thresholds of the nodes and to develop an effective learning algorithm. His work forms the basis for nearly all later work on

node					number of
2	3	4	5	6	functions
no	no	no	no	no	24
no	no	no	yes	no	8
no	yes	no	no	yes	8
yes	no	yes	no	no	8
no	yes	no	yes	yes	16
yes	no	yes	yes	no	16
no	yes	no	yes	yes	8
yes	no	yes	yes	no	8
yes	yes	yes	no	yes	16
yes	yes	yes	no	yes	8
yes	yes	yes	yes	yes	136
					256

Table 2.6: Summary of the fault tolerance of the complete 3-input perceptron of figure 2.22. It shows how many functions can tolerate a single fault in the network.

$f_2(\ldots,x_i,\ldots)$ to $f_2(\ldots,\bar{x}_i,\ldots)$). These changes cancel each other.

Such operations are not always possible; the set ϕ must be closed under negation of the output or inputs, which is indeed the case for threshold gates or ULM nodes. Hence, most perceptrons are trivially redundant. Such redundancy, however, does not contribute to fault recovery. If a fault occurs, these negation operations cannot be used to reconfigure the network.

Recognition Device") was built [Alek83a]. Their design has been used successfully for recognizing facial pictures projected onto a 512x512 array of photocells. A general overview of the work by this research group can be found in [Alek83b].

The work of Armstrong at the University of Montreal in Canada uses 2-input incomplete nodes [Arms79]. A physical implementation of their work is described in [Arms71, Arms76]; it was used for LANDSAT image processing [Arms78]. The major goal of this work is the demonstration of a feasible learning strategy [Arms79], and we will return to it in chapter 5.

2.4.2. Network Options

2.4.2.1. Interconnection complexity

The interconnections in a perceptron can be either open-loop or closed-loop. *Open-loop* perceptrons are combinational logic networks. In addition to most of the original perceptron research and our work at UCLA, other open-loop perceptrons are for example the *committee machine* studied by Takiyama [Taki78, Taki81] and some of the work by Aleksander and Armstrong.

Cross-coupled and *back-coupled* systems are sequential in nature and are harder to analyze formally. The work in this area is, as a result, less formal and consists of only a few isolated examples. No unifying theory exists. The *cognitrons* proposed by Fukushima [Fuku75, Fuku80, Miya84] are

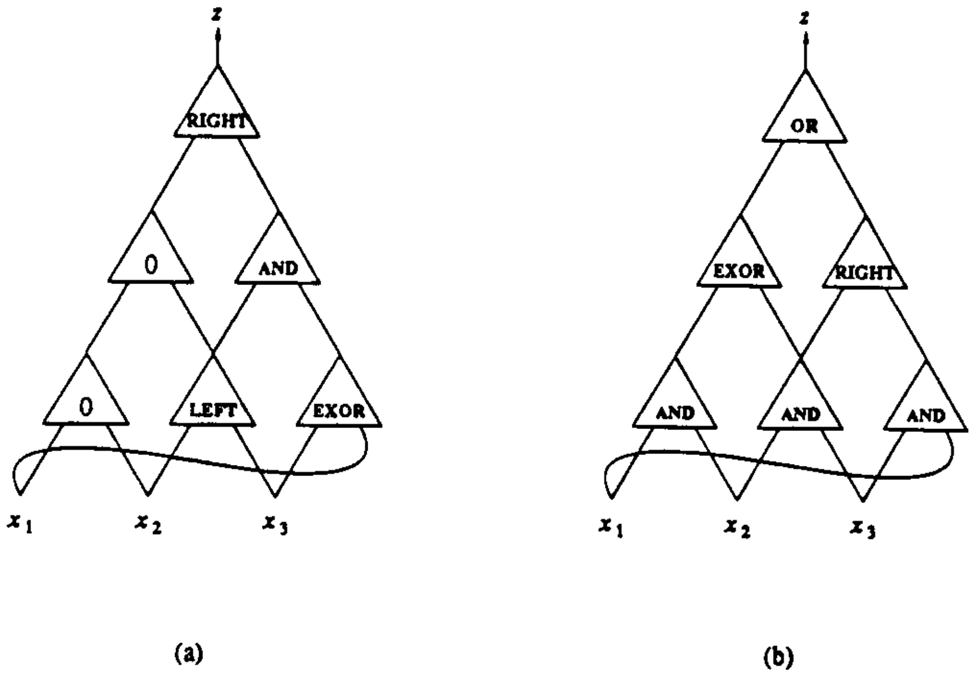


Figure 2.24: Example of nontrivial redundancy in a perceptron.

2.3.3. Summary

The perceptrons discussed in this section are simple, yet they embody the two major properties of perceptrons.

The first property is the multilayered nature of the network. Dividing a network function among multiple layers is difficult. Single-layer systems, on the other hand, always have a straightforward solution. Gaining insight into the topic of multilayered functional flexibility is the first goal of our research.

The second property is the fan-out interconnections. A disjunctive binary tree (figure 2.21) does not have this characteristic, and such a network interconnection scheme. A thorough understanding of the cause of this stability is important, not only in the context of automata theory, but also in the context of brain modeling. The brain is also capable of stable behavior even with imperfections in its underlying neural interconnections. It is an intriguing question why and how such random networks can exhibit characteristics that are useful or exhibit capabilities for such complicated functions as pattern recognition or associative recall.

b. Structured interconnections

In addition to connecting threshold gates in a random fashion, Rosenblatt also proposed to structure the network interconnections with a specific application in mind. The goal is to obtain a hierarchy of desiied feature detectors by observing a set of rules for interconnecting the nodes in different layers. The interconnections then result in a given application-dependent set of low-level or high-level features. A typical example of a modern structured perceptron is the *cognitron* proposed by Fukushima [Fuku75, Fuku80, Miya84].

The perceptron interconnections can also be structured with a different goal in mind, namely its physical implementation. Issues regarding the implementation of a perceptron may at times introduce more important constraints than feature-dependent issues. Regularity, for instance, is an important implementation constraint. Rectangular arrays are examples of networks with a regular interconnection pattern [Hopf82, Cruz83].

neural networks and learning systems.

Around the same time, Widrow used the same concept of an adjustable threshold gate in his systems called *Adaline* and *Madaline* [Widr60, Widr62]. A prototype was built using an electrochemical technology. His work, however, was not as general in scope as the perceptron research.

Most of the neural modeling research that followed is based on a threshold gate model. However, none of the proposed systems have been built in hardware; the *simple perceptron* and the *Madaline* remain the only physical implementations. On the other hand, all the function-set models reviewed in the next section form the basis of a hardware implementation. The threshold gate model, or even more complex analog models, are undoubtedly important for theoretical studies, but for practical implementations with current technology a digital approach is necessary.

2.4.1.2. Function-set model

The function-set model is a more practical and more powerful model for perceptrons, yet it has received little attention. Besides our group at UCLA, only two research groups have worked with such a model and both have practical implementations.

The group of Aleksander at Brunel University in the UK has worked with a digital model since 1968 [Alek68a, Alek70b, Alek71]. Recently they concentrated on practical implementations with RAMs [Alek79, Ston85]. A prototype system called *WISARD* ("Wilkie, Stonham, and Aleksander's the threshold gates in the bottom layer are adjusted; other nodes are fixed. In the work by Aleksander [Alek70b, Alek79] only the bottom layer consists of RAMs; the output node is fixed to an AND function, an OR function, or a voting function.

The work by Barto (University of Massachusetts) on the *associative search network* is also restricted to a single layer of threshold gates [Bart81a, Bart81b]. However, the output of each threshold gate is an output of the perceptron and all the threshold gates are independent of each other.

c. Multilayer perceptrons

Perceptrons with adjustable nodes in more than one layer are the most difficult to analyze. In the extreme case *all* the nodes in *all* the layers of the network are adjustable. Some preliminary learning schemes for multilayered perceptrons were proposed [Rose62b, Widr62, Widr64]. Other preliminary work was done by for instance Bobrowski [Bobr78], Fukushima [Fuku75, Fuku80, Fuku84], Barto [Bart82b], and Armstrong [Arms79]. In our work at UCLA it is our explicit goal to study only multilayered perçeptrons.

cross-coupled perceptrons similar to Rosenblatt's proposals [Rose60b]. Inhibitory cross-connections in the layers improve the efficiency of the system. [Fuku84, Miya84] also include back-coupling. The work by Hinton [Hint81] and much of Aleksander's work [Alek68b, Alek70b, Alek84b, Alek85] interconnects the output of the system to its inputs. Rectangular arrays of mutually communicating nodes (similar to cellular automata) are treated in [Wils76, Wils80, Hopf82, Cruz83]. The work on the Boltzmann machine also uses a rectangular array, but it is unique in its learning capabilities (see chapter 5) [Hint84, Hint85].

2.4.2.2. Interconnection topology

It is almost impossible to find two independent research groups studying exactly the same perceptron interconnection topology; every group has its own typical structure. These differences make it difficult to compare and integrate results obtained by different people.

The possible interconnection topologies found in the literature can be classified into two groups: random or structured.

a. Random interconnections

Generating the interconnections in a random fashion is the simplest but least efficient option. For instance, [Kauf70] and [Atla81] examine the limit cycles of randomly constructed networks of Boolean nodes. They show that these cycles are sometimes surprisingly independent of the particular other rule-based applications. This chapter discusses the class of combinational rule-based systems and shows how perceptrons can be used as embedded systems in such applications.

Section 3.1 gives a brief overview of rule-based systems. Section 3.2 introduces combinational rule-based systems and their implementation with perceptrons. An overview of possible interpretations of the output of a perceptron in this context follows in section 3.3. Conclusions regarding this application domain are presented in section 3.4.

3.1. RULE-BASED INFERENCE SYSTEMS

The term *rule-based inference system* represents a broad class of artificial intelligence (AI) systems, namely those that use a set of rules and a set of facts to arrive at conclusions. Other names are *rule-based deduction systems*, *knowledge-based systems*, and *expert systems*. The latter term has a meaning in AI that is somewhat incompatible with what we discuss in this chapter. For example, the user interface (the *consultation session*) is of central importance in an expert system, whereas in this chapter we assume that there is no user involved, and therefore no consultation session is needed. Names such as *production system* or *pattern directed inference system* are also related to rule-based inference systems, but imply a slightly different programming strategy. However, the result is the same, namely a decision derived from a set of facts and rules. The relationship between rule-based inference systems and perceptrons as studied in this chapter therefore applies to a general class of decision-making systems.

2.4.2.3. Number of adjustable nodes

The number of adjustable nodes in a perceptron is important not only because it determines the functional capabilities of the perceptron (the completeness factor), but also because it influences to a large extent the complexity of the algorithms necessary to control the functional changes. Simplified networks may have trivial programming or learning procedures; more general systems are more complex to control.

Based on this observation we divide the flexibility of perceptrons into three classes, listed here in order of increasing complexity.

a. Single-node perceptrons

In a *simple perceptron* only one threshold gate is involved in the functional adjustment. The same holds for the *Adaline,* which consists of only a single threshold gate, introduced by Widrow [Widr60]. Another example of a single-node perceptron is [Bart82a].

b. Single-layer perceptrons

Extensive work has been done on perceptrons with a single layer of adjustable nodes. A historical example is the network of *Adalines,* called *Madaline,* proposed by Widrow [Widr62, Widr64]. A number of *Adalines* (threshold gates) are connected to a set of inputs, and their outputs are connected to a voting gate. More recent examples are the *committee machine* [Taki78] and the *two-level committee machine* [Taki81]. In both cases only system the data base would consist of the set of data relating to one specific patient, for instance lab test results, symptoms, intermediate conclusions, etc.

b. Rule base

The rule base is sometimes called the *knowledge base.* It contains the generic "expertise" of the system, the "knowledge," expressed as a set of **if-then** causal relations. The rule base is a set of rules applicable to a class of problems. The rules are *case-independent,* but constitute *domain-specific knowledge.* In medical diagnosis a rule would for example express a causal relationship between symptoms and diseases.

c. Inference engine

The purpose of the third component, sometimes called the *control structure,* is to integrate the rules and data in a way that simulates expert reasoning in the solution of a problem. The function of the inference engine is not only *case-independent* but also *domain-independent.* It can be used for different sets of rules belonging to different application domains. It implements a certain strategy for using the rules and the data, based on notions of implication, truth-maintenance, probability theory, etc. The inference engine represents *strategy-dependent meta-knowledge.* One inference engine will not be appropriate for all possible rule-based inference systems, but only for a class of similar problem domains. Different classes of problem domains may require different strategies of reasoning.

CHAPTER 3

COMBINATIONAL RULE-BASED SYSTEMS

Two-dimensional digital *pattern classification* is the typical application domain for perceptrons. Tutorial treatments of perceptrons are found in textbooks on pattern classification [Duda73, Kova80, Bow84]. Under a more general description, however, the application domain must be extended to encompass any problem expressible as a Boolean function. This chapter shows that the applications for perceptrons include other well known domains and, in particular, an important subset of the class of *rule-based inference systems*.

Perceptrons are networks of fixed topology consisting of nodes with adjustable logic. Because of the concurrency, processing is extremely fast and therefore suited to real-time applications. Perceptrons are comparable in speed to ordinary combinational circuits, but do not need rewiring to adjust to a range of functions. The function of a perceptron can be modified without changing its physical structure.

Perceptrons are therefore useful in applications that require fast and modifiable decision-making logic. Examples include aircraft or satellite control systems, intensive-care units, monitoring and alarm systems, and many

As mentioned earlier, a typical expert system will, for convenience, have other components to deal with the user interface (not shown in the figure). A *knowledge acquisition* component facilitates the interface between the expert and the rule base. A *consultation* component facilitates the interface between the user and the data base. Other possible components include an *explanation system*, a *debugging system*, etc. The core of the rule-based inference system, however, consists of the three basic components described earlier. These components form the minimal configuration of any rule-based inference system.

3.1.2. Development and Use

A rule-based system has two distinct phases. In one phase the rule base is being created or modified; in the other phase these rules are applied to the data base. These two phases can be, and typically will be, alternating as the rule base is being designed. The two phases are described below.

a. Development of the rule base

This process is called the *knowledge acquisition session* in expert systems. In this phase the expert is the only human involved. The expert is the source of the case-independent domain-specific knowledge, expressed as a set of rules (see figure 3.2.a). These rules are entered into the rule base, or existing rules are modified or deleted.

We refer to for example [Forg81, Brow85] for a detailed treatment of production systems, and to [Wate78] for a treatment of pattern-directed inference systems. The principles of rule-based inference systems are discussed in detail in for example [Nils80, Haye83]. A theoretical treatment of the process of deriving conclusions from the set of facts and rules is given in [Kowa79].

This section reviews the principles of rule-based inference systems as an introduction to the later discussion on the relationship between these systems and Boolean logic. First we discuss their structure and knowledge hierarchy. A treatment of how they are developed and used follows. Finally, a few possible formalisms for expressing the rules are reviewed.

3.1.1. Structure

A rule-based inference system consists of three distinct components: a data base, a rule base, and an inference engine. Other components may be added for convenience or to improve the user interface, but these additions are not part of the core of the system.

a. Data base

The data base of a rule-based inference system is sometimes called the *working memory* in the expert system literature. The data base consists of a set of facts, intermediate results, and conclusions about one specific problem. It is therefore *case-specific data*. For example, in a medical diagnosis expert

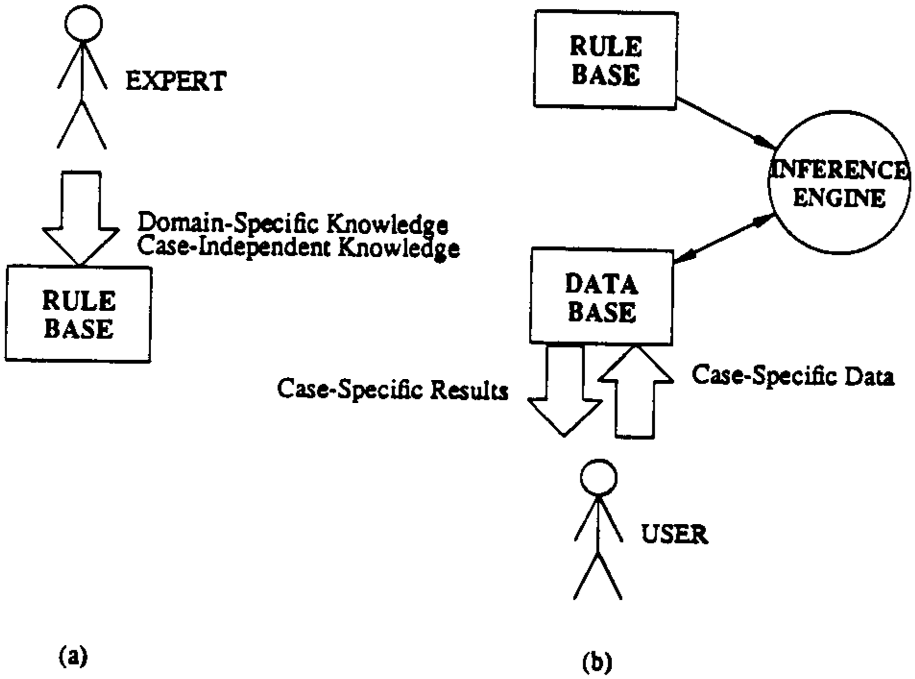


Figure 3.2: Schematic representation of the two separate phases in a rule-based inference system. (a) Development. (b) Application.

$$valve\text{-}open \textbf{ and } pump\text{-}on \rightarrow pressure$$

b. Predicate logic

Predicate logic is a more general formalism because it uses variables, universal quantifiers, and existential quantifiers. Such rules allow statements about the truth or falsity of a whole class of propositions. For example:

$$\forall x,y \,:\, \exists z \,:\, parent(x,z) \textbf{ and } parent(y,z) \textbf{ and } male(x) \rightarrow brother(y,x)$$

represents the rule that for all x, y, if there exists a z such that x and y have

A schematic representation of these three components is shown in figure 3.1. It also shows the two humans who interact with the system. On the one hand, the *expert* provides the rules for the rule base; on the other hand, the *user* is the source of case-specific information and is also the person who wants the case-specific results.

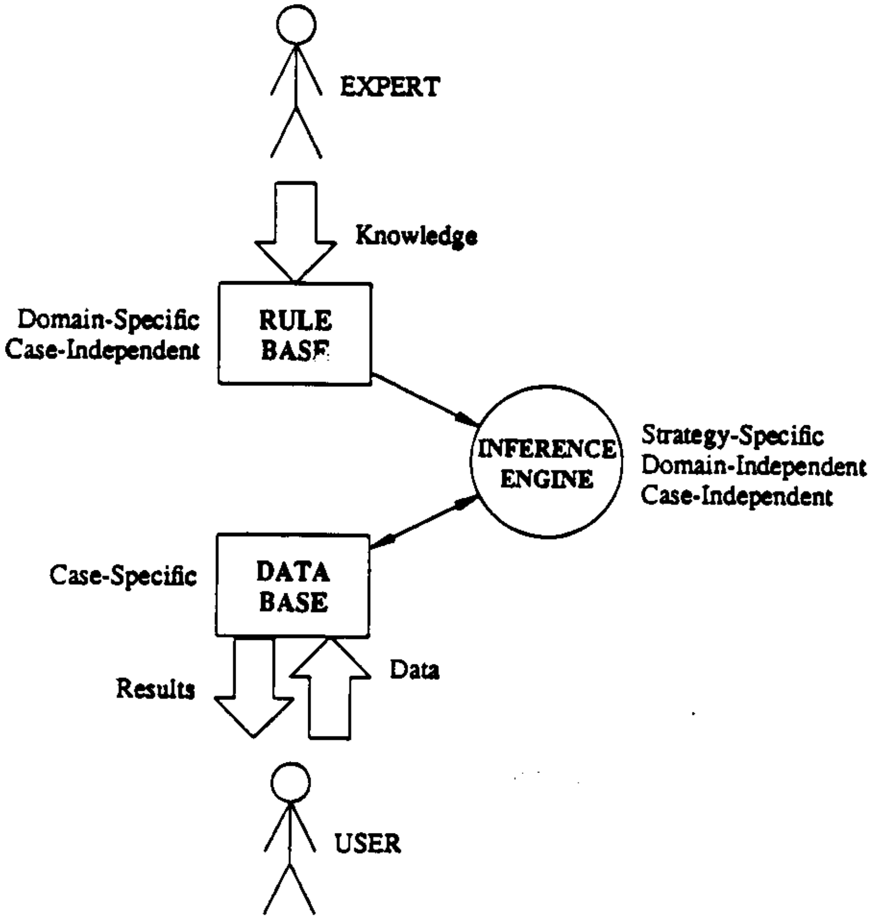


Figure 3.1: Schematic representation of the three components of a rule-based inference system. Two humans interact with the system: the expert and the user.

available. The environment is therefore entirely combinational, hence the name *combinational rule-based systems*. Such systems can be used as *embedded* components in a real-time environment.

A definition and description of this class of rule-based systems is given first. Next it is shown how the consequent of a rule can be expressed as a Boolean function of the antecedent propositions. A study of the integration of multiple rules into one Boolean function follows. An example concludes this section.

3.2.1. Definition and Description

We define the *data base* of a combinational rule-based system to be a set of *d* propositions:

$$\mathrm{D} = \{p_i \mid i=1, \ldots, d\}$$

Some of the propositions have a known truth value (*true* or *false*), either because they were asserted by sensors, or because they were derived from these propositions by the available rules. The remaining propositions have an *unknown* truth value, also called a *don't know* value. The possible values associated with each proposition in the data base are thus {0,1,?}, where "?" represents a *don't know* value. We therefore use *tri-valued* logic.

The complete truth tables (including the *don't know* value) for a conjunction and disjunction in tri-valued logic are given in table 3.1 [Turn84]. A conjunction is *false* if one of its propositions is *false;* it is *true* only if *all* the propositions are *true;* in all other cases it is *unknown.* The complementary observations hold for the disjunction. A disjunction is *true* if one of its terms is *true;* it is *false* only if *all* the propositions are *false;* in all other cases it is *unknown.*

p_1	p_2	$p_1 p_2$	p_1+p_2
0	0	0	0
0	1	0	1
0	?	0	?
1	0	0	1
1	1	1	1
1	?	?	1
?	0	0	?
?	1	?	1
?	?	?	?

Table 3.1: Truth tables for the conjunction and disjunction in tri-valued logic.

The rule base of a combinational rule-based system is a set of r rules R_i:

$$\mathsf{R} = \{R_i \mid i=1, \ldots, r\}$$

Each of these rules has an antecedent and a consequent:

$$R_i : f_i(p_{i,1}, p_{i,2}, \ldots, p_{i,k_i}) \rightarrow (p_{i,0})^{\delta_i}$$

$\delta_i \in \{0,1\}$; $(p)^0$ denotes $\bar{p}$ and $(p)^1$ denotes p. $p_{i,j} \in \mathsf{D}$, $i=1, \ldots, r$, $j=1, \ldots, k_i$. f_i is a Boolean function of the variables $p_{i,1}, \ldots, p_{i,k_i}$. If $f_i(\mathrm{p}_i) = f_i(p_{i,1}, \ldots, p_{i,k_i}) = 1$, then the rule R_i *applies* to the data base; its antecedent $f_i(\mathrm{p}_i)$ is *matched* by the data base. This rule is in a more general form than is typically found in expert systems, since f_i can be any Boolean function and not only a conjunction. In our treatment we assume that all rules with the same consequent are lumped together into one rule. This is done by combining the antecedent functions as a disjunction. Rules with the same proposition but a different sign δ_i in their consequents cannot be grouped. For example, the following four rules:

$$p_1 p_2 \rightarrow p_0$$

$$p_2 \bar{p}_3 p_4 \rightarrow \bar{p}_0$$

$$\bar{p}_2 p_3 \rightarrow p_0$$

$$\bar{p}_4 \rightarrow \bar{p}_0$$

can be grouped into two rules:

$$p_1 p_2 + \bar{p}_2 p_3 \rightarrow p_0$$

$$p_2 \bar{p}_3 p_4 + \bar{p}_4 \rightarrow \bar{p}_0$$

We also assume that the rule base does not contain loops or circular reasoning. For example $p_1 \rightarrow p_2$ combined with $p_2 \rightarrow p_1$ is not allowed.

The propositions p_i that represent the real-time measurements do not appear as the consequent of any rule in the rule base. Let us assume that n such propositions exist and that they appear first in the data base, labeled $p_1, \ldots, p_n$. Assume further that the goal proposition is the last proposition, labeled p_d. The propositions $p_1, \ldots, p_n$ are continuously assigned a value 0 or 1 by the sensors. When the inferencing starts, the remainder of the data base (including p_d) is assigned a *don't know* value. The inference engine applies a chain of rules to the data base with the goal of assigning 0 or 1 to p_d.

Since the set of external propositions $\{p_1, \ldots, p_n\}$ and the goal proposition p_d are known a priori, the partial truth table of the output p_d as a function of the input propositions $\{p_1, \ldots, p_n\}$ can be generated. This is achieved by considering all possible input assignments and recording for each assignment the decision arrived at by the inference engine. Given a certain input assignment, the antecedent of a number of rules is matched and the inference engine produces the truth values of their consequents. These conclusions may cause other rules to apply, and so on. Eventually, when no new antecedents are matched, the truth value of the proposition p_d can be observed.

Combinational rule-based systems are conceptually simple, but the implementation possibilities for the inference process have been overlooked with few exceptions. For instance, Helly describes an implementation of malfunction procedure logic on programmable-logic arrays [Hell84]. The rule base of such systems can be implemented in hardware as a combinational circuit. The rule base is prestructured a priori and transformed it into a Boolean function. The inferencing then reduces from a sequential application of rules to an input-to-output asynchronous data flow. An explicit data base is not needed. If implemented on a functionally adjustable combinational circuit, for instance a perceptron, then changes in the rule base can be achieved by modifying the functionality of the hardware. Strategies for doing this will be discussed in chapter 4. This implementation is schematically represented in figure 3.3.

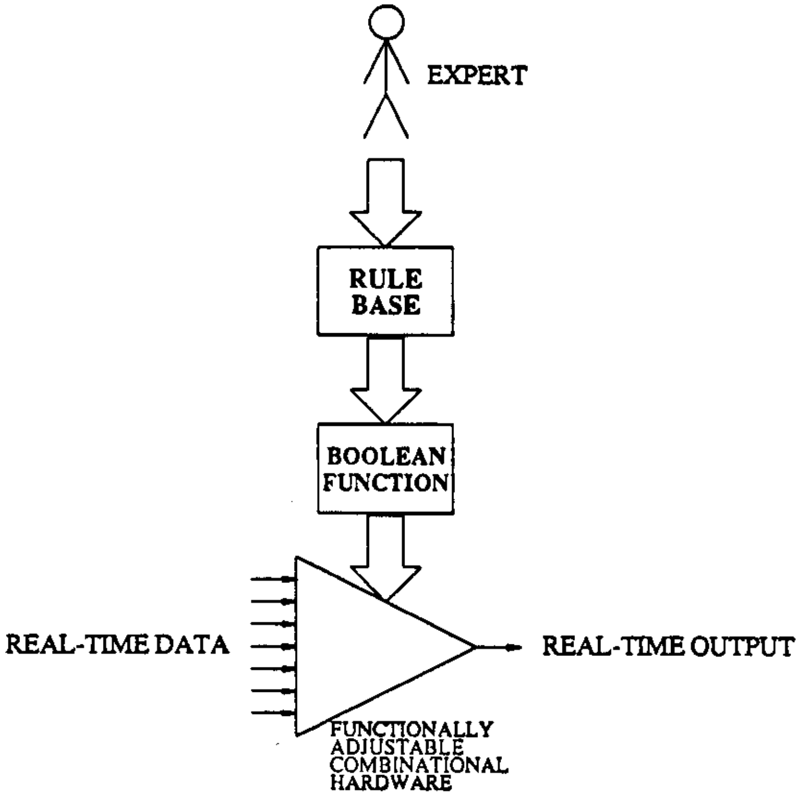


Figure 3.3: Hardware implementation of a combinational rule-based system.

3.2.2. Propositional-Logic Rules and Boolean Functions

Both a propositional-logic rule and a Boolean function express the truth or falsity of one proposition in terms of others.

Consider for instance a positive rule, which contains a non-negated consequent:

$$f_i(\mathbf{p}_i) \rightarrow p_{i,0}$$

The meaning of this rule is that the truth of $f_i(\mathbf{p}_i)$ *implies* the truth of $p_{i,0}$. In other words, if the propositions $p_{i,1}, \ldots, p_{i,k_i}$ are such that $f_i(\mathbf{p}_i)$=1, then it follows that $p_{i,0}$=1:

$$\textbf{if } f_i(\mathbf{p}_i)\mathbf{=1} \textbf{ then } p_{i,0}\mathbf{=1}$$

This rule does *not* mean:

$$\textbf{if } f_i(\mathbf{p}_i)\mathbf{=0} \textbf{ then } p_{i,0}\mathbf{=0}$$

A positive rule cannot be used to make conclusions about the *falsity* of $p_{i,0}$. If the antecedent $f_i(\mathbf{p}_i)$ is *unknown* then the conclusion is also *unknown:* p_d=?.

Direct hardware implementation of this rule requires expressing the consequent $p_{i,0}$ as a Boolean function of the antecedents $p_{i,1}, \ldots, p_{i,k_i}$:

$$p_{i,0} = f_i(\mathbf{p}_i)$$

The equality in this expression has a stronger meaning than an implication, for the former is a *bi-implication.* The equality would allow a conclusion regarding $p_{i,0}$ both if $f_i(\mathbf{p}_i)$=1 and if $f_i(\mathbf{p}_i)$=0.

Similarly, a negative rule is one where the consequent contains a negated proposition:

$$f_j(\mathbf{p}_j) \rightarrow \bar{p}_{j,0}$$

Again, the rule has no valid conclusion if the antecedent is *false*, and therefore it cannot be used to make conclusions about the ***truth*** of $p_{j,0}$. By contrast, the equality

$$\bar{p}_{j,0} = f_j(\mathbf{p}_j)$$

represents a bi-implication and allows a conclusion regarding $p_{j,0}$ both if $f_j(\mathbf{p}_j)$=1 and if $f_j(\mathbf{p}_j)$=0.

In summary, rules do not allow conclusions about both the truth *and* falsity of a proposition. If a rule is implemented as a Boolean function and its output is 1 then this implies the truth or the falsity of the consequent, depending on whether it represents a positive or negative rule. If the output is 0 then no conclusion can be made.

3.2.3. Integration of Rules

This section examines the relationshipship between the reasoning mechanism in an inference engine and the derivation of the Boolean function that expresses the truth value of a conclusion in terms of the known facts. The exhaustive procedure described in section 3.2.1 is a possible but inefficient algorithm for deriving the Boolean function equivalent to a complete rule base. In this section it is shown that substituting two Boolean functions is similar to linking two rules. By repeating such substitutions the global Boolean function that expresses the conclusion of the rule base can be obtained in a more efficient way.

Consider a rule base with two rules R_i and R_j:

$$R_i : f_i(\mathbf{p}_i) \rightarrow (p_{i,0})^{\delta_i}$$

$$R_j : f_j(\mathbf{p}_j) \rightarrow (p_{j,0})^{\delta_j}$$

Assume $p_{j,0}$ to be a proposition that appears in the antecedent of the first rule, meaning that for some l: $p_{j,0} \equiv p_{i,l}$. Assume also that the function f_i is in disjunctive normal form (a disjunction of conjunctions). Obtain the function $g_i(\mathbf{p})$ by ;eplacing every occurrence of $(p_{j,0})^{\delta_j} \equiv (p_{i,l})^{\delta_j}$ in f_i by $f_j(\mathbf{p}_j)$. This operation is similar to substitution except that $\overline{f}_j(\mathbf{p}_j)$ does not replace $(p_{j,0})^{1-\delta_j}$. In other words, a proposition and its negation are treated as two different propositions.

THEOREM 3.1: $(p_{i,0})^{\delta_j}$ can be proven *true* using R_i and R_j if and only if $g_i(\mathbf{p})=1$.

PROOF: If the antecedent of rule R_j is matched then it follows that $(p_{j,0})^{\delta_j}=1$, and this knowledge can be used in the antecedent of rule R_i. The implication and the bi-implication are now equivalent. Hence, the function $f_j(\mathbf{p}_j)$ can replace the occurrence of $(p_{j,0})^{\delta_j}$ in the antecedent function $f_i(\mathbf{p}_i)$ of rule R_i. In other words, the substitution of the Boolean function is equivalent to the linking of rules.

On the other hand, if $f_j(\mathbf{p}_j)=0$ then the implication has a different meaning than the Boolean equivalence, and R_j cannot be treated as a Boolean function assignment. Rule R_j would conclude nothing regarding $(p_{j,0})^{\delta_j}$ whereas a Boolean function would assign $(p_{j,0})^{\delta_j}$=0. Therefore, if $(p_{j,0})^{\delta_j}$ is replaced by $f_j(\mathbf{p}_j)$ then a *don't know* value is replaced by a 0 in the antecedent of R_i. What is the effect of this change? Could it result in a *true* value when it is not justified? Three cases are possible:

1. If $f_i(\mathbf{p}_i)$=0 for a particular set of propositions in which $p_{i,l}=p_{j,0}=?$, then changing the value of $p_{j,0}$ from a ? to a 0 cannot change the value of $f_i(\mathbf{p}_i)$. Indeed, since f_i is in disjunctive normal form, all its terms, which are conjunctions, must be 0 in order for $f_i(\mathbf{p}_i)$ to be 0. This means that each term has at least one proposition with a value 0 (which cannot be $p_{j,0}$), and this value is unaffected by the change in the value of $p_{j,0}$. The antecedent $f_i(\mathbf{p}_i)$ thus remains 0. Therefore, both R_i and its functional equivalent conclude *don't know.*

2. If $f_i(\mathbf{p}_i)$=1 then changing $(p_{j,0})^{\delta_j}$ from ? to 0 cannot affect the value of $f_i(\mathbf{p}_i)$. Again, $f_i(\mathbf{p}_i)$=1 occurs only if at least one term of the disjunction has a value 1. In this term, all the propositions must be 1 (implying that $p_{j,0}$ cannot be in this term). Hence, the antecedent $f_i(\mathbf{p}_i)$ remains 1. Both R_i and its functional equivalent conclude *yes.*

3. If $f_i(\mathbf{p}_i)$=? then rule R_i does not apply and the conclusion for $(p_{i,0})^{\delta_i}$ is *don't know.* If the antecedent of R_i becomes 0 as a result of changing $p_{j,0}$ from ? to 0, then R_i still does not apply and no incorrect conclu-

sions are derived. A change such that $f_i(\mathbf{p}_i)$ becomes 1, however, is not allowed because this would lead to a conclusion $(p_{j,0})^{\delta_j}=1$, and this is not justified. However, since $f_i(\mathbf{p}_i)=?$ all terms of the disjunction are either ? or 0. With the change of $(p_{j,0})^{\delta_j}$ from ? to 0, the terms that are 0 remain 0. The terms that are ? could become 0 but not 1 because only $(p_{j,0})^{\delta_j}$ and not $(p_{j,0})^{1-\delta_j}$ is replaced. Hence, both the rule and the Boolean function conclude *don't know.*

In summary, after the substitution process the Boolean function will be 1 if and only if the rules indeed allow a positive conclusion for $(p_{i,0})^{\delta_i}$; the function value will be 0 when no conclusion is possible. □

Substitutions can therefore be done for all the rules in the rule base, provided a proposition and its negation are treated as separate propositions. If one starts with the goal proposition p_d and substitutes propositions as specified by the rules, then the result is a Boolean function F^+ that expresses the truth value of p_d in terms of all the external propositions $p_1, \ldots, p_n$. Starting the substitution procedure with $\bar{p}_d$ gives a Boolean function $F^-(p_1, \ldots, p_n)$. The truth of $F^+(p_1, \ldots, p_n)$ implies the truth of p_d; the truth of $F^-(p_1, \ldots, p_n)$ implies the falsity of p_d. But the falsity of F^+ and F^- does not allow any conclusion. In summary, by a process of repeated substitution one arrives at one or two global Boolean functions that represent the knowledge in the given rule base.

3.2.4. Example

We now present a didactical example of a combinational rule-based system. This example will be used throughout the remainder of this chapter. It is extremely simple, yet it illustrates the issues that occur in any rule-based system of this form.

Consider an environment where a hydraulic system must be monitored. The system consists of a collection of pipes, pumps, valves, turbines, etc. Consider a subsystem with a primary and secondary circuit, a valve controlling each circuit, and a pump feeding both circuits. The following propositions describe the state of the physical system (their value is monitored by input sensors):

p_1=*pump-on*: the pump is on and is generating pressure

p_2=*prim-valve-open*: the valve between the pump and the primary circuit is open

p_3=*sec-valve-open*: the valve between the pump and the secondary circuit is open

Intermediate propositions can be deduced from these measurements:

p_4=*prim-pressure*: the fluid in the primary circuit is under pressure

p_5=*sec-pressure*: the fluid in the secondary circuit is under pressure

p_6=*interconnected*: the primary and the secondary circuit are interconnected

The requirement for this system is to keep either circuit pressurized, but not both, in which case it is safe to turn on a certain turbine. The goal proposition is:

p_7=*turbine-safe*: it is safe to have the turbine on

If this goal proposition is *false* then the status is ''it is unsafe to have the turbine on.'' In a real system, this status may trigger the action of turning the turbine off. We refer to figure 3.4 for a representation of the problem.

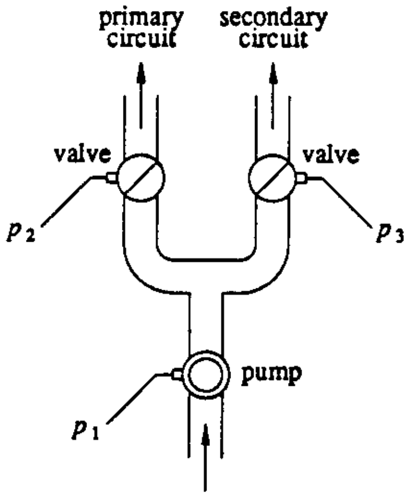


Figure 3.4: Representation of the problem for the example combinational rule-based system.

The knowledge regarding the safety of operation of the turbine can be expressed as a set of rules:

Rule 1. *pump-on* and *prim-valve-open* → *prim-pressure*

Rule 2. *pump-on* and *sec-valve-open* → *sec-pressure*

Rule 3. *prim-valve-open* and *sec-valve-open* → *interconnected*

Rule 4. not *pump-on* → not *prim-pressure*

Rule 5. not *pump-on* → not *sec-pressure*

Rule 6. not *prim-valve-open* → not *prim-pressure*

Rule 7. not *prim-valve-open* → not *interconnected*

Rule 8. not *sec-valve-open* → not *sec-pressure*

Rule 9. not *sec-valve-open* → not *interconnected*

Rule 10. *prim-pressure* and not *sec-pressure* and not *interconnected* → *turbine-safe*

Rule 11. *sec-pressure* and not *prim-pressure* and not *interconnected* → *turbine-safe*

Rule 12. *interconnected* → not *turbine-safe*

These rules are specified by an engineer with expert knowledge regarding the operation of the system. This person also selects the intermediate propositions and decides on their importance. There is a priori no guarantee that the set of rules can deduce the goal proposition in all possible cases, or that the rules are not contradictory.

Using the symbolic propositions $p_1, \ldots, p_7$ the rule base can be expressed as follows:

[illegible]

ssed as a Boole

[illegible]

Rule 1. $p_1 p_2 \rightarrow p_4$

Rule 2. $p_1 p_3 \rightarrow p_5$

Rule 3. $p_2 p_3 \rightarrow p_6$

Rule 4. $\bar{p}_1 \rightarrow \bar{p}_4$

Rule 5. $\bar{p}_1 \rightarrow \bar{p}_5$

Rule 6. $\bar{p}_2 \rightarrow \bar{p}_4$

Rule 7. $\bar{p}_2 \rightarrow \bar{p}_6$

Rule 8. $\bar{p}_3 \rightarrow \bar{p}_5$

Rule 9. $\bar{p}_3 \rightarrow \bar{p}_6$

Rule 10. $p_4 \bar{p}_5 \bar{p}_6 \rightarrow p_7$

Rule 11. $p_5 \bar{p}_4 \bar{p}_6 \rightarrow p_7$

Rule 12. $p_6 \rightarrow \bar{p}_7$

The partial truth table that represents this rule base can be derived as follows. Let us assume that both valves are closed and the pump is off. The rules 4 through 9 allow a conclusion that the propositions *prim-pressure, sec-pressure,* and *interconnected* are all *false,* but no conclusion regarding the proposition *turbine-safe* is possible. Hence, the conclusion is *don't know.* When both valves are open, but the pump is off, rule 3 concludes that the two circuits are interconnected, and using rule 12 this implies that *turbine-safe* is *false.* If on the other hand the pump is on and only the primary valve is open, then rules 1, 8, 9, and 10 imply that *turbine-safe* is *true.*

This analysis can be repeated for all possible combinations of the input propositions $p_1 = \textit{pump-on}$, $p_2 = \textit{prim-valve-open}$, and $p_3 = \textit{sec-valve-open}$. The result is summarized in table 3.2.

p_1	p_2	p_3	p_7	Rules
0	0	0	*don't know*	-
0	0	1	*don't know*	-
0	1	0	*don't know*	-
0	1	1	*false*	3,12
1	0	0	*don't know*	-
1	0	1	*true*	2,6,7,11
1	1	0	*true*	1,8,9,10
1	1	1	*false*	3,12

Table 3.2: Summary of the truth values of the goal proposition p_7 for all possible values of the input propositions and the rules that were used to arrive at the conclusion.

Alternatively, the Boolean functions that express the truth value of p_7 can be derived by a substitution process. Combining rules 10 and 11 into one rule results in:

$$p_4\bar{p}_5\bar{p}_6+p_5\bar{p}_4\bar{p}_6\rightarrow p_7$$

Similarly, the rules 4 and 6, 5 and 8, and 7 and 9 can be grouped into:

$$\bar{p}_1+\bar{p}_2\rightarrow\bar{p}_4$$

$$\bar{p}_1+\bar{p}_3\rightarrow\bar{p}_5$$

$$\bar{p}_2+\bar{p}_3\rightarrow\bar{p}_6$$

The goal proposition is expressed as a Boolean function:

$$p_7=p_4\bar{p}_5\bar{p}_6+\bar{p}_4p_5\bar{p}_6$$

and by substitution it follows:

$$p_7 = p_4 \bar{p}_5 \bar{p}_6 + \bar{p}_4 p_5 \bar{p}_6$$

$$= (p_1 p_2)(\bar{p}_1 + \bar{p}_3)(\bar{p}_2 + \bar{p}_3) + (\bar{p}_1 + \bar{p}_2)(p_1 p_3)(\bar{p}_2 + \bar{p}_3)$$

$$= p_1 p_2 \bar{p}_3 + p_1 \bar{p}_2 p_3$$

$$= F^+(p_1, p_2, p_3)$$

Starting from $\bar{p}_7$ results in

$$\bar{p}_7 = p_6 = p_2 p_3 = F^-(p_1, p_2, p_3)$$

3.3. INTERPRETATION

An inference engine integrates the rules in a rule base to derive conclusions. A combinational rule-based system derives conclusions by applying one or more Boolean functions to the external propositions. Below is an overview of the possible interpretations of the output of these Boolean functions.

3.3.1. Positive Rules Only

If the rule base contains only positive rules then it can be represented by a single Boolean function:

$$p_d = F^+(p_1, \ldots, p_n) = F^+(\mathbf{p})$$

The output of this function can take 2 values: $F^+(\mathbf{p})=0$ or $F^+(\mathbf{p})=1$. An output 1 implies that p_d is *true.* For these particular values of the external propositions, the given rules allow a *proof* that p_d is *true.* An output $F^+(\mathbf{p})=0$

means that a positive conclusion is not possible, but neither is a negative one since no negative rules are available. The conclusion is therefore *don't know*. The possible conclusions can be represented using a Venn diagram (figure 3.5). The same figure also shows schematically the combinational circuit implementing that Boolean function. The interpretation of the output of F^+ is summarized in table 3.3.

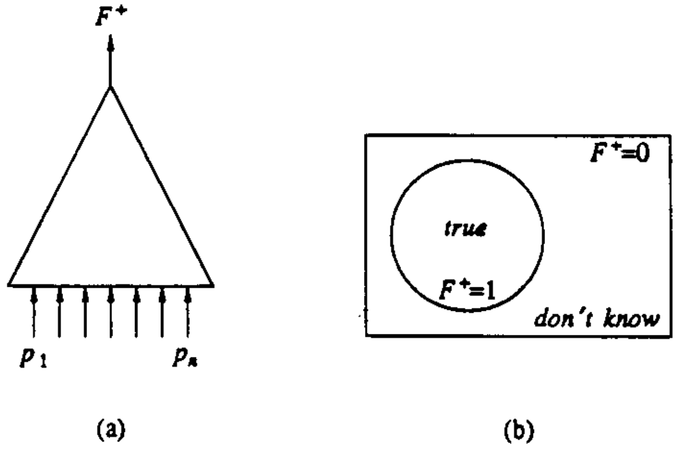


Figure 3.5: A propositional-logic rule base with only positive rules expressed as a Boolean function. (a) Combinational circuit implementing the Boolean function. (b) Venn diagram representing the conclusions.

In practical systems the inference mechanism will usually assume truth or falsity for p_d even if F^+(p)=0. In other words, the inference engine will assume a specific conclusion even when there is no formal basis for doing so. By far the most common strategy (if not the only one) in present-day rule-based inference systems is to assume p_d to be *false* if F^+(p)=0. Typical

F^+(p)	p_d
0	*don't know*
1	*true*

Table 3.3: Summary of the conclusions regarding p_d when a rule base is expressed as a Boolean function $F^+(p_1, \ldots, p_n)$.

examples of languages for rule-based inference systems based on this approach are Prolog [Cloc81] or OPS5 [Forg81]. In such languages, if a proposition cannot be proven then it is assumed to be *false*. This approach is equivalent to assuming each rule to represent a ***bi-implication*** instead of an implication [Aida83, Jaff83].

EXAMPLE 3.1: The following positive rules are extracted from the rule base given in section 3.2.4:

Rule 1. $p_1 p_2 \rightarrow p_4$

Rule 2. $p_1 p_3 \rightarrow p_5$

Rule 3. $p_2 p_3 \rightarrow p_6$

Rule 10′. $p_4 \rightarrow p_7$

Rule 11′. $p_5 \rightarrow p_7$

The resulting truth table for this modified rule base is given in table 3.4. Compare this with table 3.2. The corresponding Boolean function is now:

$$F^+(p_1,p_2,p_3) = p_1 p_2 + p_1 p_3$$

A possible implementation of F^+ on the perceptron of figure 2.17 is given in figure 3.6. □

p_1	p_2	p_3	$F^+(p_1,p_2,p_3)$	p_7
0	0	0	0	*don't know*
0	0	1	0	*don't know*
0	1	0	0	*don't know*
0	1	1	0	*don't know*
1	0	0	0	*don't know*
1	0	1	1	*true*
1	1	0	1	*true*
1	1	1	1	*true*

Table 3.4: Truth table for F^+ and the interpretation for the goal proposition p_7 for the case of a rule base with only positive rules.

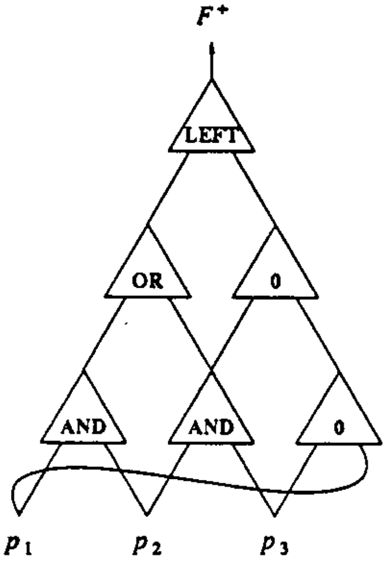


Figure 3.6: A possible implementation on a UCLA perceptron of the example rule base containing positive rules only.

3.3.2. Positive and Negative Rules

If the rule base contains both positive and negative rules then the substitution process results in two separate Boolean functions:

$$p_d = F^+(p_1, \ldots, p_n)$$

$$\bar{p}_d = F^-(p_1, \ldots, p_n)$$

This is shown schematically in figure 3.7.a. To follow our previous assumption of single-output perceptrons, we consider the two functions to be separate. Typically, however, the set of rules leading to a positive and a negative conclusion are not entirely separate and the system implementing these two functions would consist of two overlapping combinational circuits. We refer to [Mart86] for an approach that reduces the combinational hardware necessary for multi-output combinational rule-based system.

Four different output combinations are possible. A Venn diagram representing these combinations is shown in figure 3.7.b. A theoretically justified conclusion for p_d is possible only if either F^+(p) or F^-(p) are 1, but not both. If F^+(p)=1 and F^-(p)=0 then applying the rules to the given data base allows a proof that p_d is *true*. Vice versa, if F^+(p)=0 and F^-(p)=1 then p_d can be proven *false*. If both F^+(p) and F^-(p) produce an output 1 for the given data then there is a conflict in the rule base. If both F^+(p) and F^-(p) are 0 then the conclusion is *don't know*. A summary of the conclusions is given in table 3.5.

If both F^+(p) and F^-(p) are 0 then in theory no conclusion is possible. In practical systems, however, the inference engine will typically still assign

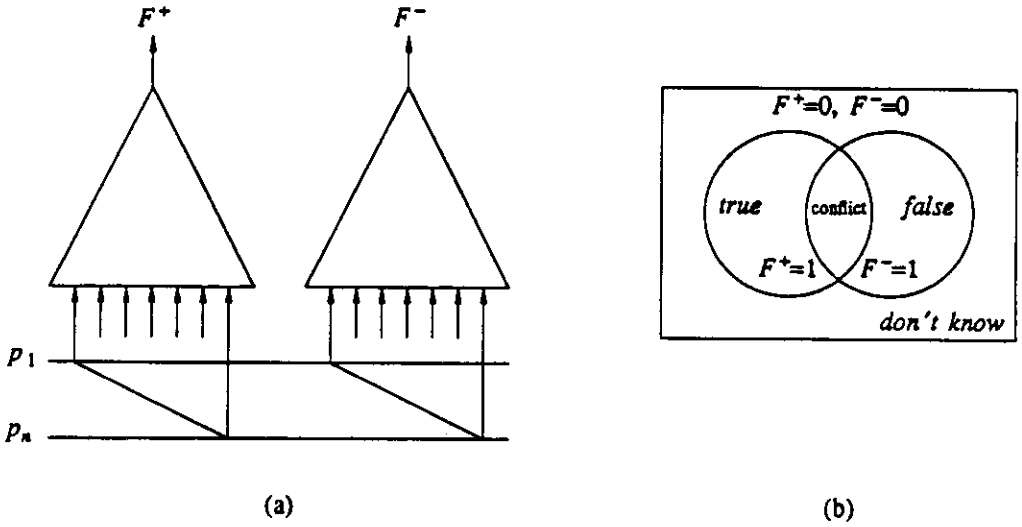


Figure 3.7: A propositional-logic rule base with positive and negative rules expressed as two Boolean functions. (a) Combinational circuits. (b) Venn diagram.

F^+(p)	F^-(p)	p_d
0	0	*don't know*
0	1	*false*
1	0	*true*
1	1	*conflict*

Table 3.5: Summary of the conclusions regarding p_d when a rule base is expressed as two Boolean functions F^+ and F^-.

a value *true* or *false* to p_d, even though there is no formal basis for doing so. For instance, it might assume p_d to be *false* if F^+(p)=0, or alternatively p_d=*true* if F^-(p)=0.

Since the rule base contains both positive and negative rules, it is possible to have conflicts in the rule base. These conflicts surface when the goal proposition p_d can be proven to be both *true* and *false,* in other words both F^+(p)=1 and F^-(p)=1. If this occurs then the inference system should alarm the expert who designed the rule base, point out the conflict, and have the rule base modified in such a way that the conflict is resolved. In the meantime no decision can be made. Practical systems will typically still assume truth or falsity if a conflict occurs, for instance by giving priority to the positive rules, or to the rules that were most recently added to the rule base.

EXAMPLE 3.2: Consider again the rule base of section 3.2.4. The two corresponding Boolean functions are:

$$F^+(p_1,p_2,p_3) = p_1 p_2 \bar{p}_3 + p_1 \bar{p}_2 p_3$$

$$F^-(p_1,p_2,p_3) = p_2 p_3$$

A summary is given in table 3.6. A possible implementation of these functions on two perceptrons is shown in figure 3.8. □

EXAMPLE 3.3: **Inconsistent rule base.** Assume rules 10 and 11 are altered as follows:

p_1	p_2	p_3	$F^+(p_1,p_2,p_3)$	$F^-(p_1,p_2,p_3)$	p_7	Rules
0	0	0	0	0	*don't know*	-
0	0	1	0	0	*don't know*	-
0	1	0	0	0	*don't know*	-
0	1	1	0	1	*false*	3,12
1	0	0	0	0	*don't know*	-
1	0	1	1	0	*true*	2,6,7,11
1	1	0	1	0	*true*	1,8,9,10
1	1	1	0	1	*false*	3,12

Table 3.6: Truth table of F^+ and F^- and the interpretation for the goal proposition p_7 for the case of a rule base with positive and negative rules.

Rule 10′. $p_4 \rightarrow p_7$

Rule 11′. $p_5 \rightarrow p_7$

In this case:

$$F^+(p_1,p_2,p_3) = p_1 p_2 + p_1 p_3$$

$$F^-(p_1,p_2,p_3) = p_2 p_3$$

and a truth table is given in table 3.7. Rules 10′ and 11′ are inconsistent. A possible implementation on perceptrons is given in figure 3.9. □

3.3.3. Complete and Consistent Rule Base

In the development of a rule base for an expert system, the goal is to improve, by a process of trial-and-error, the rule base and strive towards a system where conflicts and *don't know* conclusions do not occur. To reduce *don't know* conclusions, the *don't know* regions in figure 3.7.b must be

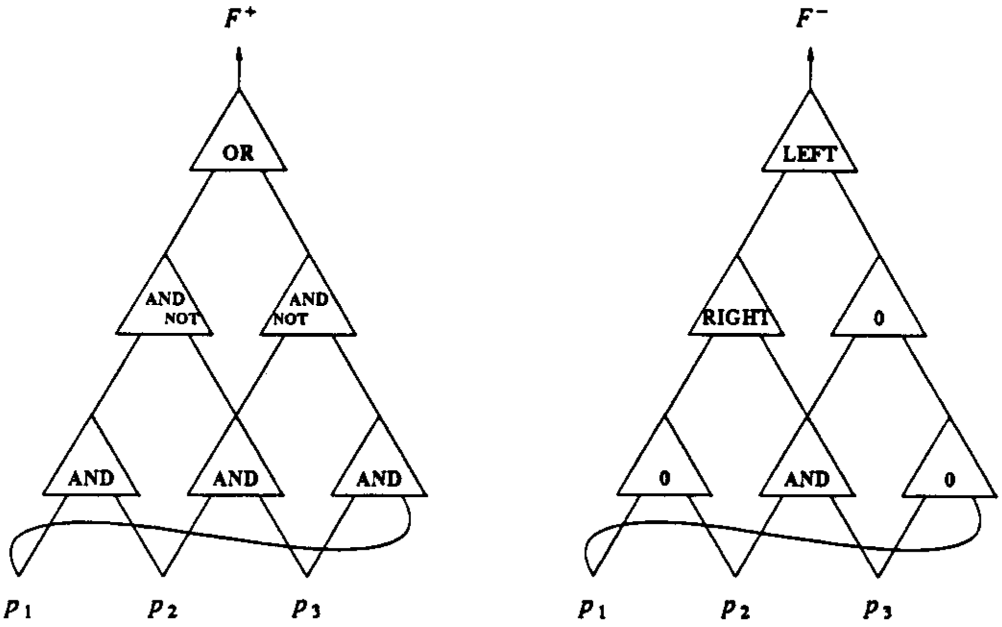


Figure 3.8: A possible implementation on two UCLA perceptrons of the example rule base containing positive and negative rules.

identified and reduced, or even eliminated if so desired. In theory, it is possible to design a *complete* rule base, although it is in practice hard to achieve. Additionally, conflict cases must be identified and eliminated. Both *don't know* and conflict cases could be found, for instance, by a simple exhaustive search in which all the possible input assignments are examined. However, reasoning about the positive and negative rules is a more efficient strategy to find these problem cases. See [Suwa84] for an example of such an approach. *Don't know* or conflicting conclusions can be reduced or eliminated by removing or modifying existing rules in the rule base, or by adding new ones. If the rule base is complete and consistent then the Venn diagram

p_1	p_2	p_3	$F^+(p_1,p_2,p_3)$	$F^-(p_1,p_2,p_3)$	p_7	Rules
0	0	0	0	0	*don't know*	-
0	0	1	0	0	*don't know*	-
0	1	0	0	0	*don't know*	-
0	1	1	0	1	*false*	3,12
1	0	0	0	0	*don't know*	-
1	0	1	1	0	*true*	2,11′
1	1	0	1	0	*true*	1,10′
1	1	1	1	1	*conflict*	1,10′/3,12

Table 3.7: Truth table for F^+ and F^- and the interpretation for the goal proposition p_7 for the case of a rule base that contains conflicts.

representing the possible decisions is as in figure 3.10. The corresponding list of possible outputs and conclusions is summarized in table 3.8.

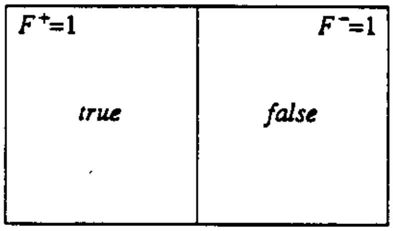


Figure 3.10: Venn diagram representing the possible decisions for an inference system with a complete and consistent rule base.

In such an inference system, F^+ and F^- are perfect complements of each other, which implies that one suffices for a decision. In other words, only the positive (or the negative) rules are needed, and a *don't know* can be assumed to be *false* (see figure 3.5 and table 3.3).

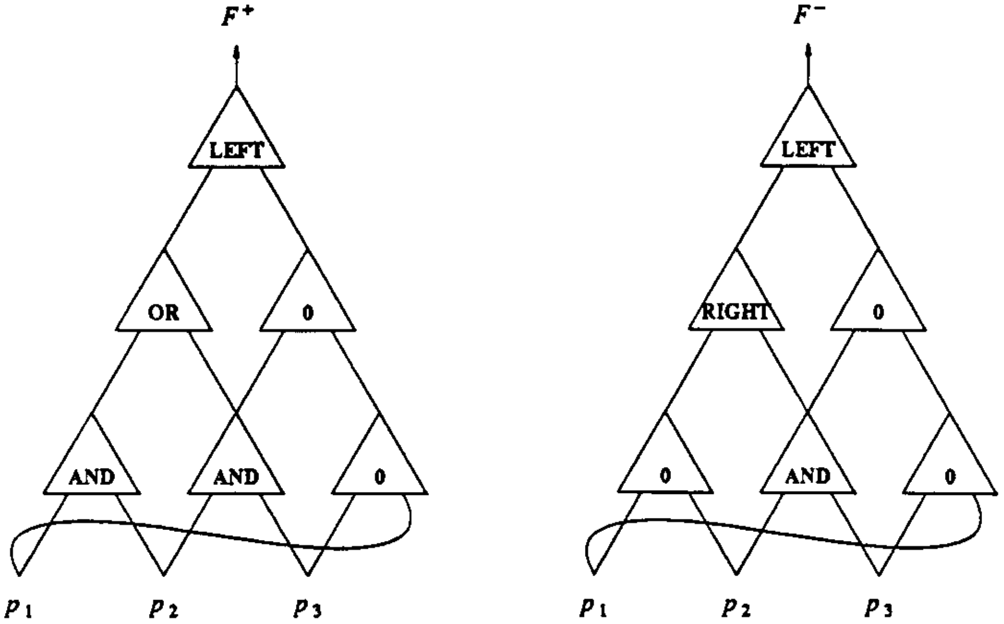


Figure 3.9: A possible implementation on perceptrons of the example rule base containing conflicts.

$F^+(\mathrm{p}_i)$	$F^-(\mathrm{p}_i)$	p_d
0	1	*false*
1	0	*true*

Table 3.8: Summary of the possible conclusions for an inference system with a complete and consistent rule base.

In the development of the rule base, the expert is constantly revising the set of rules and doing experiments to try them out. In this phase there is no guarantee that the rule base is complete, nor that it is consistent. A decomposition of the knowledge into positive and negative rules is therefore desirable. Therefore, practical rule-based inference systems should process both positive and negative rules. A hardware implementation, for instance with perceptrons, then requires two separate circuits.

EXAMPLE 3.4: Complete and consistent rule base. Assume the following rule is added to the rule base of section 3.2.4:

Rule 13. $\bar{p}_4\bar{p}_5 \rightarrow \bar{p}_7$

The Boolean functions now become:

$$F^+(p_1,p_2,p_3) = p_1 p_2 \bar{p}_3 + p_1 \bar{p}_2 p_3$$

$$F^-(p_1,p_2,p_3) = \bar{p}_4 \bar{p}_5 + p_6 = \bar{p}_1 + \bar{p}_2 \bar{p}_3 + p_2 p_3 = \bar{F}^+(p_1,p_2,p_3)$$

with a summarizing truth table given in table 3.9. A possible implementation on perceptrons is given in figure 3.11. The two perceptrons are identical except for an inversion of polarity in the top node. □

p_1	p_2	p_3	$F^+(p_1,p_2,p_3)$	$F^-(p_1,p_2,p_3)$	p_7	Rules
0	0	0	0	1	*false*	4,5,13
0	0	1	0	1	*false*	4,5,13
0	1	0	0	1	*false*	4,5,13
0	1	1	0	1	*false*	3,12
1	0	0	0	1	*false*	6,8,13
1	0	1	1	0	*true*	2,6,7,11
1	1	0	1	0	*true*	1,8,9,10
1	1	1	0	1	*false*	3,12

Table 3.9: Truth table of F^+ and F^- and the interpretation for the goal proposition p_7 for the case of a complete and consistent rule base.

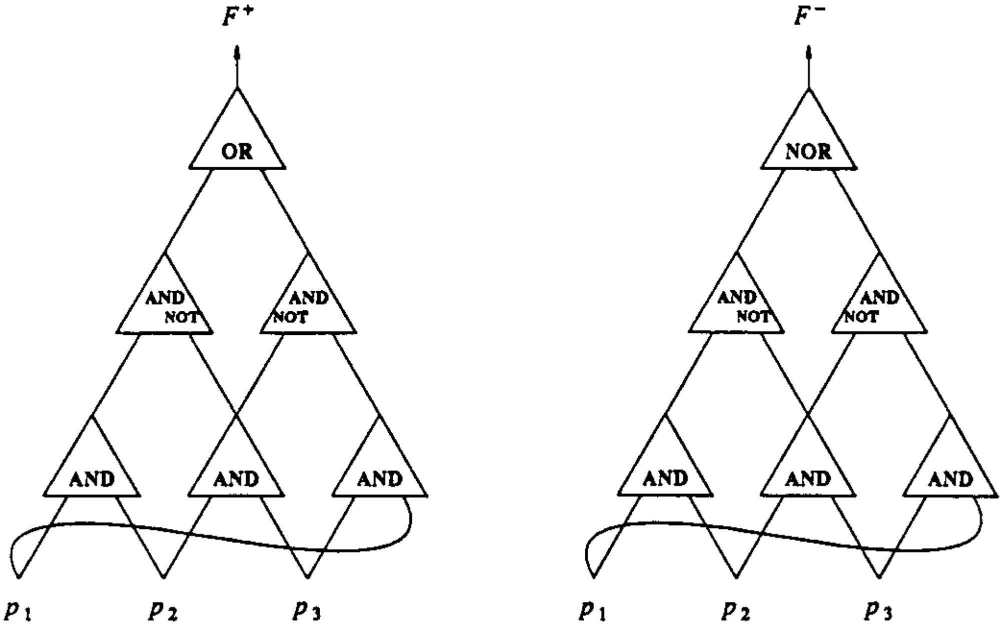


Figure 3.11: A possible implementation of the complete and consistent rule base on perceptrons.

3.4. CONCLUSIONS

The discussions in this chapter have shown that Boolean functions are useful tools for implementing a class of rule-based inference systems. Combinational rule-based systems address problems of diagnosis, interpretation, monitoring, control, and so on. These applications are indeed problems of pattern recognition or combinational logic. By contrast, expert systems for design, planning, computer aided instruction, which are sequential in their human interface, would not benefit from being implemented with combinational circuits.

The requirement that the rule base contains rules in propositional logic is too strong and some rule bases using predicate logic are also combinational. The transformation of the rule base into a Boolean function is more elaborate for predicate logic because unification must be used. Future research should explore in detail this larger class of combinational rule-based systems.

In the context of combinational rule-based systems, perceptrons are always dealing with completely specified Boolean functions. When implementing a complete and consistent rule base, the corresponding Boolean function may be either F^+ or F^-. Both are complements of each other and both are completely specified. If the rule base is not complete or consistent then two Boolean functions and therefore two perceptrons are necessary to represent the rule base. Both functions F^+ and F^- are again completely specified.

CHAPTER 4

DECOMPOSITION OF BOOLEAN FUNCTIONS ON PERCEPTRONS

This chapter discusses one aspect of the control of a perceptron, namely the selection of the functions at each node such that the perceptron implements a given network function. This problem of implementing a given function on a network is similar in purpose to *programming* a sequential computer. However, we view it as the *decomposition* of one large Boolean function into a set of smaller Boolean functions.

The decomposition problem is a degenerated form of learning. The desired network function is assumed to be specified in advance and incremental changes in the function are not considered. Changing the network function in any way will require starting over to find the nodal functions that accomplish the new function. More flexible forms of learning will be discussed in chapter 5.

The central topic in the decomposition problem is the *assignment of functional responsibility:* what parts of the given global function must be assigned to which nodes of the network? We apply the theory of decomposition of Boolean functions to this problem and show that in certain simple networks the solution to the decomposition problem can be obtained in a straightforward fashion. In more general perceptrons, however, a search strategy is required and a direct assignment of responsibility is not possible. We develop a heuristic criterion that reduces the amount of search necessary to find a decomposition.

Section 4.1 presents introductory issues such as definitions, problem description, implementation possibilities, and literature review. Section 4.2 presents a special-purpose solution to the decomposition problem, a solution that applies only to specific perceptron structures. Section 4.3 is a treatment of the concepts that will be used in the decomposition algorithm. The algorithm itself is developed in section 4.4, including a program that implements the method. Section 4.5 presents some conclusions.

4.1. INTRODUCTION

4.1.1. Definitions

Perceptrons consist of polyfunctional combinational nodes. In this chapter, the nodes of the network are assumed to be all *identical* because it simplifies the description. However, the principles and methods outlined here are generalizable to heterogeneous networks.

Each node i has k inputs $\mathbf{x}_i=(x_{i,1}, \ldots, x_{i,k})$ and its output z_i is a Boolean function of its inputs:

$$z_i = f_i(\mathbf{x}_i) = f_i(x_{i,1}, \ldots, x_{i,k})$$

The nodal functions are adjustable independently and input or output values do not influence these functions directly. In other words, the function f_i can be any one of a set of possible functions:

$$\phi = \{f_1, \ldots, f_p\}$$

The node of a perceptron is formally defined to be a pair:

$$\mathsf{N} \equiv (k, \phi)$$

where k is the *number of inputs* and ϕ is the *functional set.*

Each input of a node is connected to the output of another node, and the interconnections form a single-output loop-free network. The interconnections are defined by a set of *node interconnections:*

$$\mathsf{I} \equiv \{(i, z_i, \mathbf{x}_i) \mid i=1, \ldots, l\}$$

where:

i is a unique label for the node;

z_i is the output label of the node;

$\mathbf{x}_i=(x_{i,1}, \ldots, x_{i,k})$ are the input labels of the node, an ordered set of either output labels of other nodes or labels of the network inputs;

l is the number of nodes in the perceptron.

Each node interconnection $(i, z_i, \mathbf{x}_i)$ specifies that the inputs of node i are connected to respectively $x_{i,1}, \ldots, x_{i,k}$, and that its output is labeled z_i. The latter can be used in other node interconnections.

The definition of an entire perceptron is a 4-tuple:

$$\mathsf{P} \equiv [\mathsf{N}, z, \mathbf{x}, \mathsf{I}]$$

where:

N is the definition of the node;

z is the network output;

$\mathbf{x}=(x_1, \ldots, x_n)$ are the network inputs;

I is the set of node interconnections.

The resulting network implements a Boolean function of n inputs:

$$z = F(\mathbf{x}) = F(x_1, \ldots, x_n)$$

This network function F is derived from the choices f_i for the functions of the constituent nodes. By changing these functions the network implements a set of different functions:

$$\Phi = \{F_1, \ldots, F_Q\}$$

Φ is not part of the definition of a perceptron, but can be derived from it.

An *assignment* of node functions to a perceptron is a mapping

$$\alpha : \{1, \ldots, l\} \rightarrow \phi : i \rightarrow f_i$$

meaning that the node labeled i is *assigned* the function $f_i \in \phi$. The corresponding network function is represented by F_α. A *partial* assignment (one in which not all elements of the domain are mapped) is a subset of a *complete* assignment.

4.1.2. Problem Description

Assume that a perceptron definition P and a Boolean function G are specified. The problem is to implement the given function with the given perceptron by finding an appropriate assignment for all the node functions. Stated differently, the problem is to decompose a Boolean function of n inputs into a set of smaller functions of k inputs each $(k<n)$ in such a way that it matches the given perceptron structure. More formally, a *decomposition problem* is defined as follows:

Given: a perceptron definition $\mathsf{P} \equiv [\mathsf{N}, z, \mathbf{x}, \mathsf{I}]$ and a goal function $G(\mathbf{x}) = G(x_1, \ldots, x_n)$,

Find: an assignment α such that $F_\alpha(\mathbf{x}) \equiv G(\mathbf{x})$.

As a secondary problem, it may be required to find different (or all) possible assignments $\{\alpha_j\}$ that implement the goal function, if more than one exists.

4.1.3. Implementation

The implementation of the decomposition schemes proposed in this chapter assumes that one *central controller*, for instance a conventional sequential computer, receives the desired network function from the user and generates the appropriate control bits for all the nodes in the perceptron (figure 4.1). This approach implies two separate phases in the operation of the system. In the first phase, the *decomposition phase*, the controller decomposes the given network function and generates the control. In the second phase, the *data processing phase*, the perceptron processes external data.

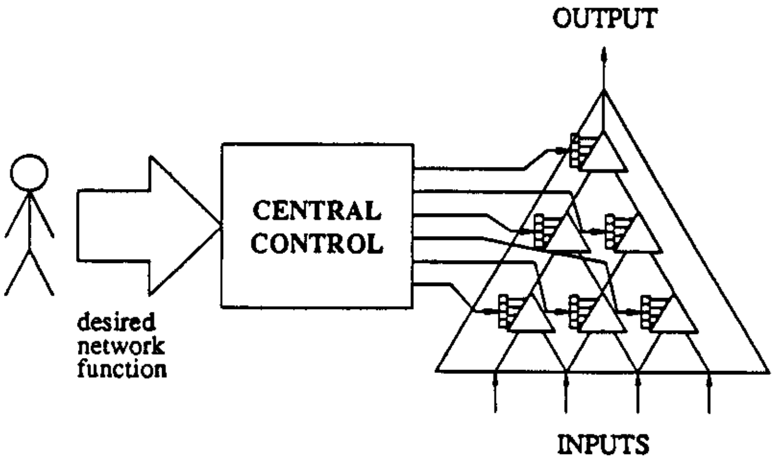


Figure 4.1: Implementation of a perceptron with a central controller.

Alternatively, in an implementation with *distributed control* each node of the perceptron has a small local controller attached to it. These controllers receive the same goal function, run autonomously, communicate only with neighboring nodes, and exchange as little information as possible. This implementation would have many practical advantages, but it is a topic for future research.

4.1.4. Literature Review

Despite the extensiveness of the perceptron literature, no general solution to the decomposition problem is available. The literature, dealing mainly with threshold gate networks, does not address the decomposition problem. Instead, it concentrates on two related topics.

Most of the threshold gate literature is in the area of learning by example; it will be reviewed in chapter 5. The decomposition problem, however, can be a first step towards understanding the problems encountered in learning by example. A formal approach to the general decomposition problem provides principles that can be used in learning by example. The usefulness of Minsky and Papert's work [Mins69], for instance, lies in their theoretical treatment of what subpart of a given task can or cannot be assigned to the R-unit of a *simple perceptron,* independent of any learning scheme. Furthermore, a strategy for learning by example, where knowledge of the goal function is distributed, incremental, and incomplete, cannot perform better than a strategy for decomposition, with centralized, a priori, and complete information.

The second part of the threshold gate literature deals with the incompleteness of threshold gates and how to synthesize a threshold gate network that implements a given Boolean function. The set of threshold functions is known to be incomplete, but deciding whether a Boolean function is a threshold function has not been solved in its generality. Examples of partial treatments of this question are [Came60, Muro61, Muro62, Sing62, Wind63, Nils65, Wind65, Yaji65, Sriv77]. With networks of threshold gates the issues become even more complex. The problem addressed in the literature on threshold gate networks is also different from ours. Instead of treating the perceptron as a given, all the literature discusses how to *synthesize* a network of threshold gates that implements *one* given Boolean function. See for example [Minn61, Miil62, Amar64, Negr64, Tohm64, Sriv78].

The little work in our context is in a digital domain, unrelated to threshold gates, and consists of the theory of decomposition of Boolean functions [Curt61, Curt63, High73, Frie75]. It also focusses on synthesizing minimal networks, but provides a set of tools and principles that can be applied to the decomposition problem. These principles will be reviewed in section 4.3.

4.2. A SPECIAL-PURPOSE SOLUTION

In this section the flexibility of a class of UCLA perceptrons is reduced by fixing the functions of certain nodes, while leaving the remaining nodes adjustable. For this special case, a solution to the decomposition problem exists, but it is limited in two respects. First, the restricted networks have adjustable nodes in the bottom layer only and the decomposition problem for such perceptrons is straightforward (the assignment of responsibility can be done in a trivial fashion). Secondly, the strategy applies only to a restricted subclass of perceptrons. In other words, this solution does not address the general issue of multilayered decomposition.

Nevertheless, we study this subset of perceptrons and the corresponding decomposition strategy because it identifies a class of complete perceptrons. More specifically, we will prove the completeness of the perceptrons of figures 2.17 and 2.20.

4.2.1. The 3-Input Network

THEOREM 4.1: Consider the 3-input perceptron of figure 2.17 with the following partial assignment (figure 4.2):

node 1: $z = f_1(z_2,z_3) = z_2+z_3$.

node 2: $z_2 = f_2(z_4,z_5) = z_4 z_5$.

node 3: $z_3 = f_3(z_5,z_6) = \bar{z}_5 z_6$.

node 5: $z_5 = f_5(x_2,x_3) = x_2 \oplus x_3$.

Call f_{ij}^4 and f_{ij}^6 the entries of the truth table of the function at the nodes 4 and 6 respectively, that is, $f_{ij}^4=f_4(i,j)$ and $f_{ij}^6=f_6(i,j)$. Let $G(x_1,x_2,x_3)$ be the goal function for the perceptron and G_{ijk} its truth table entries, that is, $G_{ijk}=G(i,j,k)$. The following assignment for f_4 and f_6 constitutes a solution to this decomposition problem:

$$f_{00}^4=G_{001}\,,\, f_{01}^4=G_{010}\,,\, f_{10}^4=G_{101}\,,\, f_{11}^4=G_{110}$$

$$f_{00}^6=G_{000}\,,\, f_{01}^6=G_{011}\,,\, f_{10}^6=G_{100}\,,\, f_{11}^6=G_{111}$$

PROOF: The given assignments imply that

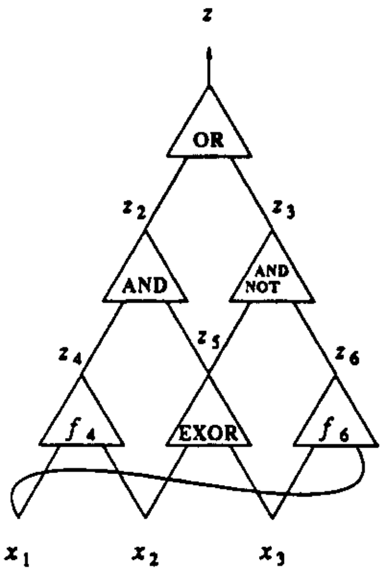


Figure 4.2: 3-input complete perceptron with a partial assignment.

$$z_4 = G_{001}\bar{x}_1\bar{x}_2 + G_{010}\bar{x}_1x_2 + G_{101}x_1\bar{x}_2 + G_{110}x_1x_2$$

$$z_6 = G_{000}\bar{x}_1\bar{x}_3 + G_{011}\bar{x}_1x_3 + G_{100}x_1\bar{x}_3 + G_{111}x_1x_3$$

and we know that

$$z_5 = x_2 \oplus x_3 = x_2\bar{x}_3 + \bar{x}_2x_3$$

Hence, by substitution:

$$z_2 = z_4z_5 = G_{001}\bar{x}_1\bar{x}_2x_3 + G_{010}\bar{x}_1x_2\bar{x}_3 + G_{101}x_1\bar{x}_2x_3 + G_{110}x_1x_2\bar{x}_3$$

$$z_3 = \bar{z}_5z_6 = G_{000}\bar{x}_1\bar{x}_2\bar{x}_3 + G_{011}\bar{x}_1x_2x_3 + G_{100}x_1\bar{x}_2\bar{x}_3 + G_{111}x_1x_2x_3$$

which makes

$$z = z_2+z_3 = G_{000}\bar{x}_1\bar{x}_2\bar{x}_3+G_{001}\bar{x}_1\bar{x}_2x_3+G_{010}\bar{x}_1x_2\bar{x}_3+G_{011}\bar{x}_1x_2x_3+$$

$$G_{100}x_1\bar{x}_2\bar{x}_3+G_{101}x_1\bar{x}_2x_3+G_{110}x_1x_2\bar{x}_3+G_{111}x_1x_2x_3$$

This shows that for any goal function G and $\forall\, x_1,x_2,x_3 : z=G(x_1,x_2,x_3)$ and hence, with this assignment: $F_\alpha(x_1,x_2,x_3) \equiv G(x_1,x_2,x_3)$. □

In this strategy, the truth table entries of the goal function are assigned to specific local truth table entries at the nodes 4 and 6. These two adjustable nodes each have 16 functions, totaling 256 different combinations (P=256). This particular perceptron can implement any of the 3-input Boolean functions ($Q=2^{2^3}$=256). It therefore follows that γ=1 and ρ=1: the perceptron is complete and nonredundant. This proves the completeness of the perceptron of figure 2.17.

Many alternative partial assignments can be derived from the one in figure 4.2 by negating inputs or by symmetry operations. An exhaustive search showed that 1024 configurations are possible. Some of them are listed in table 4.1; for a complete list we refer to [Moor85a]. Different partial assignments would require different permutation strategies.

4.2.2. The n-Input Network

Completeness proofs and decomposition strategies of the type presented above can be obtained for any number of inputs by applying the same principle in a recursive way. Figure 4.3 summarizes the approach for n inputs. The left subnetwork is complete and takes all inputs except x_k; the right sub-

node 1	node 2	node 3	node 5
$f_1(z_2,z_3)=$	$f_2(z_4,z_5)=$	$f_3(z_5,z_6)=$	$f_5(x_2,x_3)=$
z_2+z_3	z_4z_5	$\bar{z}_5z_6$	$x_2\oplus x_3$
$z_2\oplus z_3$	$\bar{z}_4z_5$	$\bar{z}_5z_6$	$x_2\otimes x_3$
z_2z_3	$z_4+\bar{z}_5$	z_5+z_6	$x_2\oplus x_3$
$z_2\otimes z_3$	$z_4+\bar{z}_5$	$z_5+\bar{z}_6$	$x_2\otimes x_3$
$z_2+\bar{z}_3$	z_4z_5	z_5+z_6	$x_2\oplus x_3$
$z_2\oplus x_3$	$z_4\bar{z}_5$	$z_5\oplus z_6$	x_2x_3
$z_2\oplus z_3$	z_4+z_5	z_6	x_2x_3
$z_2\oplus z_3$	z_4z_5	$z_5\otimes z_6$	x_2+x_3
$z_2\oplus z_3$	$z_4\oplus z_5$	$\bar{z}_5z_6$	$x_2\oplus x_3$
$z_2\oplus z_3$	z_4	z_5+z_6	$x_2\oplus x_3$

Table 4.1: A list of possible partial assignments for the complete 3-input perceptron structure.

network is also complete and takes all inputs except x_l. The center node is an *exclusive-or* gate and takes only x_k and x_l as inputs. In this network the network function of n variables is decomposed into two functions of $n-1$ variables. These two functions are implemented by the two $(n-1)$-input complete subnetworks; their outputs are combined by the four remaining nodes.

THEOREM 4.2: Consider the perceptron of figure 4.3 with goal function $G(x_1, \ldots, x_n)$. Call the functions of the two subnetworks $f_l(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n)$ and $f_r(x_1, \ldots, x_{l-1}, x_{l+1}, \ldots, x_n)$. Label the truth table entries as follows:

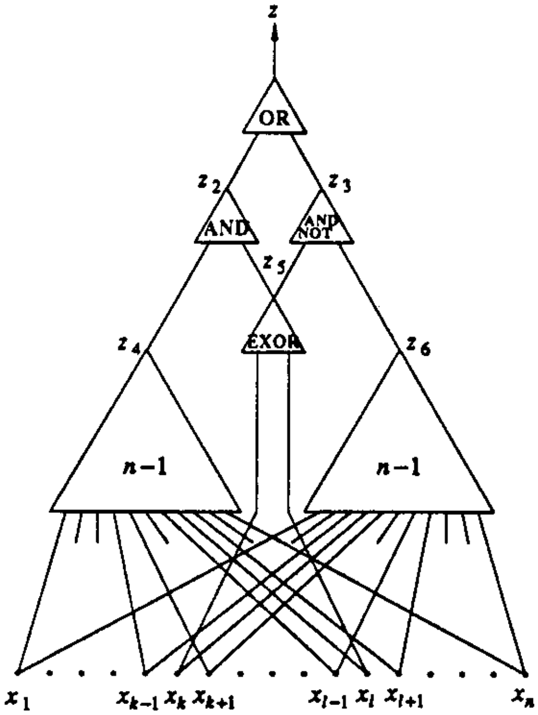


Figure 4.3: Recursive design of a complete n-input perceptron.

$$G_{i_1 i_2 \cdots i_n} = G(i_1, i_2, \ldots, i_n)$$

$$f^l_{i_1 \cdots i_{k-1} i_{k+1} \cdots i_n} = f_l(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$$

$$f^r_{i_1 \cdots i_{l-1} i_{l+1} \cdots i_n} = f_r(i_1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n)$$

with i_j=0 or 1. The following assignment for f_l and f_r implements the goal function G:

$$f^{l}_{i_1 \cdots i_{k-1} i_{k+1} \cdots i_l \cdots i_n} = G_{i_1 \cdots i_{k-1} \bar{i}_l i_{k+1} \cdots i_l \cdots i_n}$$

$$f^{r}_{i_1 \cdots i_k \cdots i_{l-1} i_{l+1} \cdots i_n} = G_{i_1 \cdots i_k \cdots i_{l-1} i_k i_{l+1} \cdots i_n}$$

PROOF: If the input to the network is $\mathbf{x}=(i_1, \ldots, i_k, \ldots, i_l, \ldots, i_n)$ then the output z becomes

$$\begin{aligned}
z &= z_2 + z_3 \\
&= z_5 z_4 + \bar{z}_5 z_6 \\
&= (i_k \oplus i_l) f_l(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n) + \overline{(i_k \oplus i_l)} f_r(i_1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \\
&= (i_k \oplus i_l) f^{l}_{i_1 \cdots i_{k-1} i_{k+1} \cdots i_n} + (i_k \otimes i_l) f^{r}_{i_1 \cdots i_{l-1} i_{l+1} \cdots i_n} \\
&= (i_k \oplus i_l) G_{i_1 \cdots i_{k-1} \bar{i}_l i_{k+1} \cdots i_l \cdots i_n} + (i_k \otimes i_l) G_{i_1 \cdots i_k \cdots i_{l-1} i_k i_{l+1} \cdots i_n} \\
&= G_{i_1 \cdots i_k \cdots i_l \cdots i_n}
\end{aligned}$$

It follows that with this assignment, for any goal function G: $F_\alpha(x_1, \ldots, x_n) \equiv G(x_1, \ldots, x_n)$. □

In the theorem above, all possible choices of $k, l \in \{1, \ldots, n\}$ $(k \neq l)$ are allowed, but most of them do not result in a network with regular topology. A simple choice is $k=n$ and $l=1$. For instance, for $n=4$, $k=n$, and $l=1$, a network results that is a substructure of the network of figure 2.20. In other words, figure 2.20 can be obtained by adding extra nodes to the 4-input structure of figure 4.3. This proves the completeness of the perceptron of figure 2.20.

4.3. BASIC PRINCIPLES OF THE DECOMPOSITION ALGORITHM

In this section the basic principles behind the decomposition strategy, to be developed in the next section, are presented. The strategy consists of the repeated application of two basic operations, namely *selection* and *reduction.* The first subsection below gives an overview of the existing theory of decomposition of Boolean functions, which forms the basic tool for the selection step. The second subsection presents the concept of reduction, used to determine the function of the remainder of the network after one or more nodes have received a functional assignment.

4.3.1. Theory of Decomposition of Boolean Functions [Curt61, Curt63, High73, Frie75]

A Boolean function $G(x_1, x_2, \ldots, x_n)$ is *decomposable* if G can be realized as a composition of functions of fewer than n variables each. Let $X = \{x_1, x_2, \ldots, x_n\}$ represent the set of input variables and assume that $A_1, A_2, \ldots, A_k$ are sets of input variables such that $\bigcup_{i=1}^{k} A_i = X$, then

$$G(X) = G'(f_1(A_1), f_2(A_2), \ldots, f_k(A_k))$$

is a decomposition of the function G. This decomposition is shown schematically in figure 4.4.

Decompositions of Boolean functions can be classified as either disjunctive or nondisjunctive. In a *disjunctive* decomposition the input variables to different functions f_i are disjoint, that is, $\forall i, j,\ i \neq j : A_i \cap A_j = \emptyset$. If, for

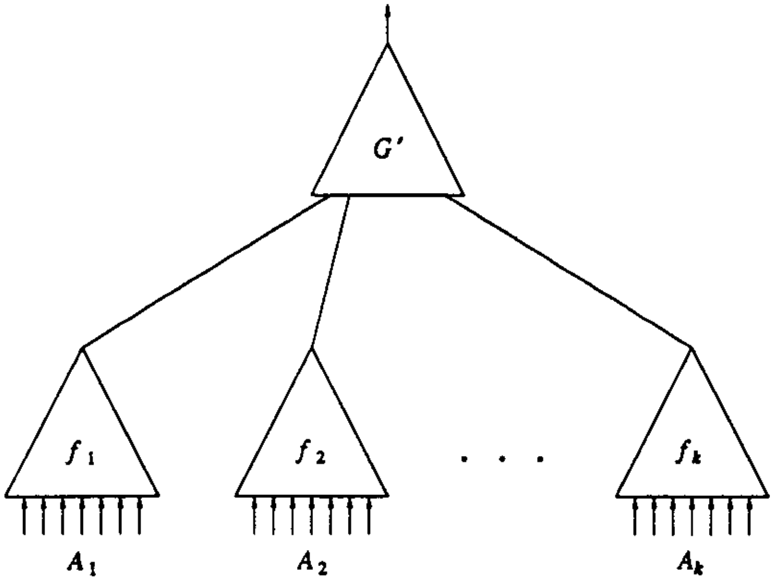


Figure 4.4: Decomposition of a Boolean function.

some i,j, $i \neq j$, $A_i \cap A_j \neq \emptyset$, the decomposition is called *nondisjunctive*.

4.3.1.1. Simple disjunctive decomposition

A simple disjunctive decomposition is a decomposition of the form $G(X)=G'(f_1(A_1),A_2)$, with $A_1 \cap A_2=\emptyset$ and $A_1 \cup A_2=X$. It can be characterized by arranging the truth table of G as in figure 4.5. The rows of the map represent entries with equal values for the variables in A_2 and the columns represent entries with equal values for the variables in the set A_1. The truth table arranged this way is called the *decomposition map* or *decomposition chart* with respect to the variable sets (A_1,A_2). If A_1 has u input variables ($|A_1|=u$) and A_2 has v input variables ($|A_2|=v$), and hence $|X|=n=u+v$, then each row of the decomposition map has 2^u entries and each column has 2^v entries.

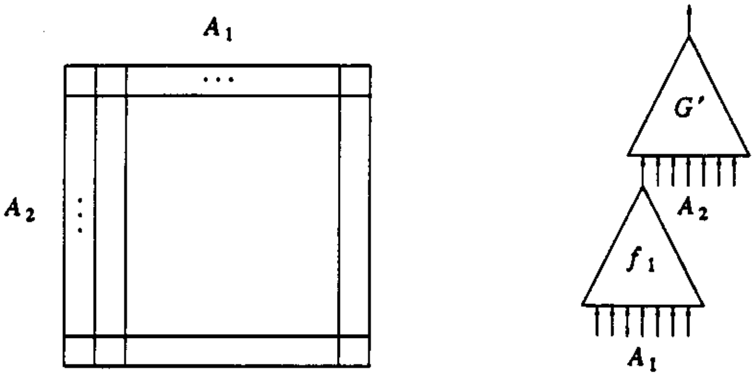
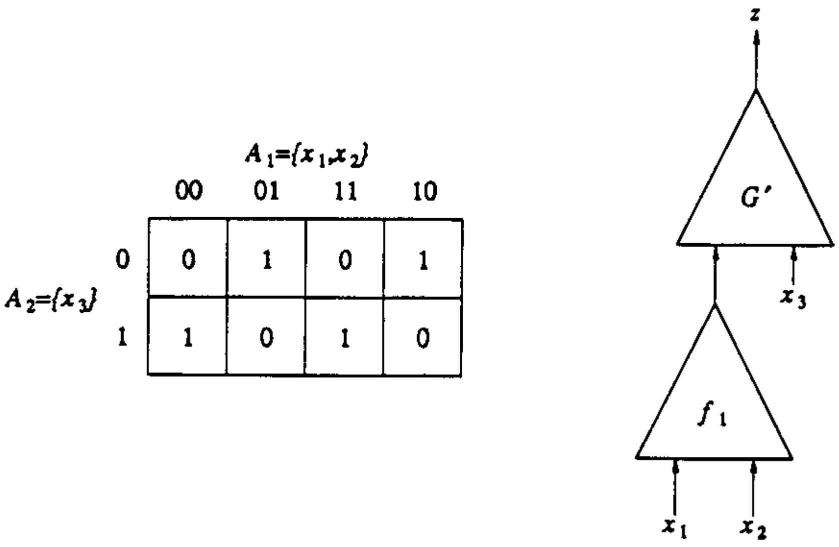


Figure 4.5: Decomposition chart for the variable sets (A_1,A_2).

THEOREM 4.3: [Frie75] A completely specified Boolean function $G(X)$ has a simple disjunctive decomposition, $G(X)=G'(f_1(A_1),A_2)$, if and only if its decomposition chart with respect to the variable sets (A_1,A_2) has at most two distinct columns (columns with different patterns of 0's and 1's). □

EXAMPLE 4.1: A simple disjunctive decomposition of the function $G(x_1,x_2,x_3) = x_1 \oplus x_2 \oplus x_3$ with respect to $A_1=\{x_1,x_2\}$ and $A_2=\{x_3\}$ is: $G(x_1,x_2,x_3) = G'(f_1(x_1,x_2),x_3)$ with $z_1=f_1(x_1,x_2) = x_1 \oplus x_2$ and $G'(z_1,x_3) = z_1 \oplus x_3$. Its decomposition chart, shown in figure 4.6, has two distinct columns, namely "01" and "10." □

EXAMPLE 4.2: In figure 4.7 $G(x_1,x_2,x_3,x_4,x_5)$ is decomposed with respect



$A_2=\{x_3\}$	$A_1=\{x_1,x_2\}$ 00	01	11	10
0	0	1	0	1
1	1	0	1	0

Figure 4.6: Example of a simple disjunctive decomposition of a Boolean function with 3 variables.

to $A_1=\{x_1,x_2,x_3\}$ and $A_2=\{x_4,x_5\}$. In this example

$$G(x_1,x_2,x_3,x_4,x_5) = x_4(\bar{x}_1x_2+\bar{x}_2x_3)+x_5(x_1x_2+\bar{x}_2\bar{x}_3)$$

$$z_1=f_1(x_1,x_2,x_3)=\bar{x}_1x_2+\bar{x}_2x_3$$

$$G'(z_1,x_4,x_5)=x_4z_1+x_5\bar{z}_1$$

Again, the decomposition chart of G has only 2 distinct columns, namely "0110" and "0011." □

If all the columns of a decomposition chart are identical then the function is independent of the variables in the set A_1. Hence, $G(X)=G'(A_2)$ and any choice for the function f_1 is valid. If the decomposition chart has two distinct columns, but the column patterns consist of all 0's or all 1's, then the

		$A_1=\{x_1,x_2,x_3\}$							
		000	001	011	010	110	111	101	100
$A_2=\{x_4,x_5\}$	00	0	0	0	0	0	0	0	0
	01	1	0	0	0	1	1	0	1
	11	1	1	1	1	1	1	1	1
	10	0	1	1	1	0	0	1	0

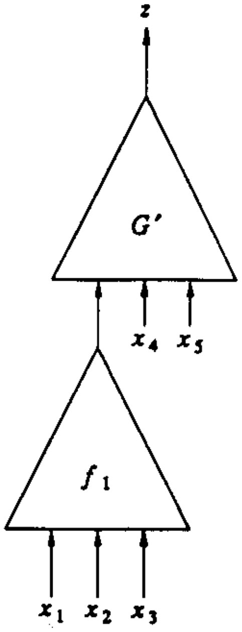


Figure 4.7: Example of a simple disjunctive decomposition of a Boolean function with 5 variables.

function is independent of the variables of the set A_2, $G(X)=G'(f_1(A_1))$, and f_1 is determined by the column differences of the decomposition chart. In a typical case the decomposition map will have 2 nontrivial columns.

The decomposition map for an incompletely specified function has *don't care* entries. Two columns (rows) of a decomposition map are *compatible* if the *don't care* entries can be assigned 0's or 1's such that the columns (rows) become identical. The decomposition of an incompletely specified Boolean function is generally not unique since different assignments to the *don't care* entries may result in different decompositions. If more than two columns of the decomposition map are mutually incompatible then a simple disjunctive decomposition with respect to the particular variable sets is not possible.

A classical problem in logic circuit design is to find the sets A_1 and A_2 such that a simple disjunctive decomposition is possible. In the decomposition problem, however, the network, and therefore the sets A_1 and A_2, are *given* and one needs to find the functions G' and f_1, if they exist.

4.3.1.2. Simple nondisjunctive decomposition

In a *nondisjunctive* decomposition the input variables of different functions f_i have some common elements. Hence, if

$$G(X) = G'(f_1(A_1), f_2(A_2), \ldots, f_k(A_k))$$

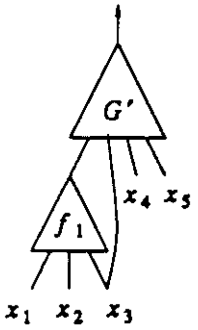
with $\bigcup_{i=1}^{k} A_i = X$ and $\exists i, j,\ i \neq j$ such that $A_i \cap A_j \neq \varnothing$, then the expression represents a nondisjunctive decomposition. A Boolean function $G(X)$ has a *simple* nondisjunctive decomposition if $G(X) = G'(f_1(A_1), A_2)$ with $A_1 \cup A_2 = X$ and $A_1 \cap A_2 \neq \varnothing$. Let $A_1 \cap A_2 = A_{12}$, $|A_{12}| = v$, $B_1 = A_1 - A_{12}$, $|B_1| = u_1$, $B_2 = A_2 - A_{12}$, $|B_2| = u_2$, and hence $u_1 + u_2 + v = n = |X|$. For each of the 2^v values for the variables in A_{12} we define a decomposition map with 2^{u_2} rows (corresponding to B_2) and 2^{u_1} columns (corresponding to B_1).

THEOREM 4.4: [Frie75] A completely specified Boolean function $G(X)$ has a simple nondisjunctive decomposition if and only if each of these 2^v maps has not more 2 distinct columns. □

EXAMPLE 4.3: In figure 4.8, $B_1 = \{x_1, x_2\}$, $B_2 = \{x_4, x_5\}$, and $A_{12} = \{x_3\}$. The functions of the two nodes are:

$$f_1(x_1,x_2,x_3)=\overline{x}_3(x_1x_2)+x_3(x_1+x_2)$$

$$G'(z_1,x_3,x_4,x_5)=\overline{x}_3[z_1\otimes(x_4+x_5)]+x_3[z_1+(x_4x_5)] \quad \square$$



		$B_1=\{x_1,x_2\}$			
		00	01	11	10
$B_2=\{x_4,x_5\}$	00	1	1	0	1
	01	0	0	1	0
	11	0	0	1	0
	10	0	0	1	0

$x_3=0$

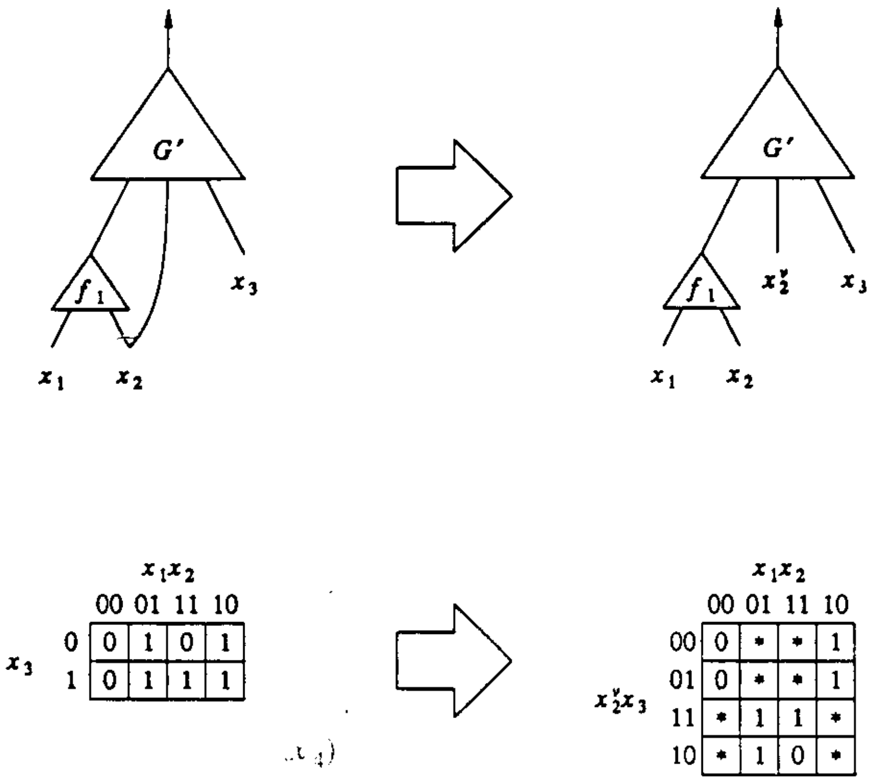
		$B_1=\{x_1,x_2\}$			
		00	01	11	10
$B_2=\{x_4,x_5\}$	00	0	1	1	1
	01	0	1	1	1
	11	1	1	1	1
	10	0	1	1	1

$x_3=1$

Figure 4.8: Example of a simple nondisjunctive decomposition of a Boolean function with 5 variables.

In our work we treat the nondisjunctive case in an alternative but equivalent way because it shows the decomposition process in a simpler way. Every nondisjunctive decomposition is converted into a disjunctive decomposition by introducing for each input $x_i \in A_{12}$ an additional *virtual* variable x_i^v. This virtual variable x_i^v must always be equal to x_i, and therefore the truth table entries with $x_i^v \neq x_i$ are *don't care.*

EXAMPLE 4.4: In figure 4.9, $B_1=\{x_1\}$, $A_{12}=\{x_2\}$, $B_2=\{x_3\}$. The introduction of x_2^v makes the network disjunctive with $A_1=\{x_1,x_2\}$ and $A_2=\{x_2^v,x_3\}$. However, since $x_2^v \equiv x_2$ half of the new decomposition chart contains *don't care* entries (represented by "*" in the figure). □



x_3 \ x_1x_2	00	01	11	10
0	0	1	0	1
1	0	1	1	1

$x_2^vx_3$ \ x_1x_2	00	01	11	10
00	0	*	*	1
01	0	*	*	1
11	*	1	1	*
10	*	1	0	*

Figure 4.9: Example showing how a nondisjunctive decomposition can be made disjunctive by introducing virtual variables.

Each virtual variable introduces *don't cares* equal in number to the size of the original truth table. The decomposition chart that results is called the *virtual* decomposition chart. The original one is called the *real* decomposition chart.

The theorem below follows directly from theorems 4.3 and 4.4; it is given here without proof.

THEOREM 4.5: A Boolean function $G(X)$ has a simple nondisjunctive decomposition if and only if its virtual decomposition chart has not more than two mutually incompatible columns. □

4.3.2. Reduction

The process called *reduction* answers the following question (figure 4.10). Suppose that a perceptron implements the function G and that one of its nodes in the bottom layer, labeled 1, implements the function f_1. What is the *residual* function G' of the remainder of the network?

In figure 4.10 G is a function of (x_1,x_2,x_3,x_4) and G' is a function of (z_1,x_2,x_3,x_4). If G' were known then G could be deduced by substituting $f_1(x_1,x_2)$ for z_1 in G':

$$G(x_1,x_2,x_3,x_4) = G'(f_1(x_1,x_2),x_2,x_3,x_4)$$

In the decomposition problem, instead of G' and f_1 being given, G and f_1 are given and G' must be deduced.

EXAMPLE 4.5: Suppose

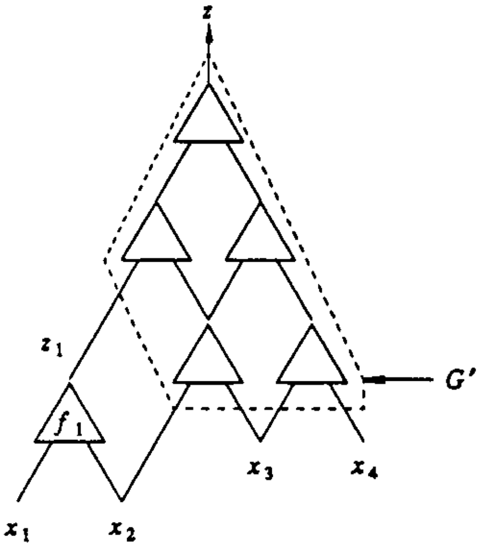


Figure 4.10: Schematic representation of reduction in a perceptron.

$$G(x_1,x_2,x_3,x_4) = (x_1+x_2)\oplus(x_3x_4)$$

and $f_1(x_1,x_2)=x_1+x_2$, then

$$G'(z_1,x_2,x_3,x_4) = z_1\oplus(x_3x_4)$$

If $f_1(x_1,x_2)=\bar{x}_1\bar{x}_2$, then

$$G'(z_1,x_2,x_3,x_4) = \bar{z}_1\oplus(x_3x_4)$$

However, if $f_1(x_1,x_2) = x_1x_2$ then reduction is *impossible* for the following reason. If the network input is (x_1,x_2,x_3,x_4) = (0,0,0,0) then the network output should be $z=G(0,0,0,0)=0$; the output of node 1 for this input is $z_1=f_1(0,0)=0$. If the network input is x=(1,0,0,0) then the network output must be $z=G(1,0,0,0)=1$; $z_1=f_1(1,0)$ is still equal to 0. What should be the truth value of G' for $(z_1,x_2,x_3,x_4) = (0,0,0,0)$? The two requirements imposed by G are conflicting. Hence, the assignment $f_1(x_1,x_2)=x_1x_2$ can never be correct for this function. □

Another important feature of reduction is that, besides sometimes being impossible, it sometimes also generates *don't care* entries in the truth table of the residual network function.

EXAMPLE 4.6: Consider figure 4.11 with $G(x_1,x_2,x_3) = (\bar{x}_1x_2)x_2x_3 = (\bar{x}_1x_2)x_3$. Assume node 1 implements $f_1(x_1,x_2) = \bar{x}_1x_2$. Is in this example $G'(z_1,x_2,x_3)=z_1x_2x_3$ or $G'(z_1,x_2,x_3)=z_1x_3$? The answer is: neither of them. Since z_1 cannot be 1 if x_2 is 0, nothing is specified regarding G' when (z_1,x_2)=(0,0), and hence G'(1,0,0) and G'(1,0,1) are *don't cares.*

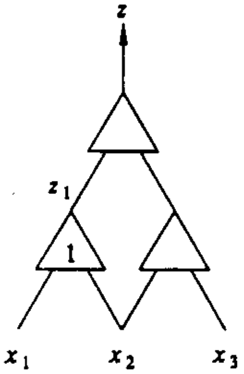
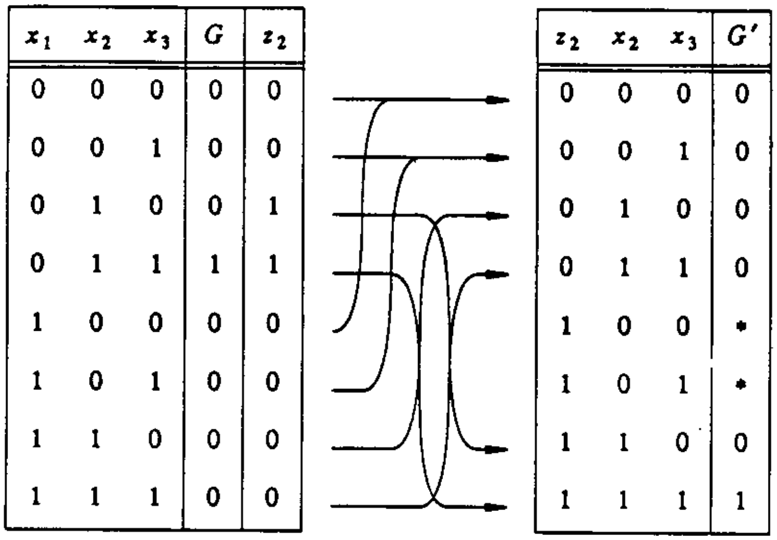


Figure 4.11: Example of reduction in a 3-input perceptron.

Figure 4.12 shows how the reduction proceeds in this example. It is a mapping from the truth table of G to the truth table of G'. Some truth table entries of G are mapped into the same entry in G'. For instance $G(0,0,0)$ and $G(1,0,0)$ are both mapped into $G'(0,0,0)$. If these entries of G were different then a conflict would occur and the reduction would not be possible. Some truth table entries of G' do not receive a value, for instance $G'(1,0,0)$ and $G'(1,0,1)$. These entries are *don't cares.* □



x_1	x_2	x_3	G	z_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

z_2	x_2	x_3	G'
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	*
1	0	1	*
1	1	0	0
1	1	1	1

Figure 4.12: Reduction as a mapping from one truth table to another.

The algorithm for the reduction process, expressed in a Pascal-like pseudo-language, is as follows:

```
function reduce($G$, $f$, conflict);
begin initialize $G'$ to all don't-care;
      conflict ← false;
      for all $s \in \{0,1\}^n$
      do begin z ← $f(s_1)$;
               if $G'(z, s_2)$ = don't-care
               then $G'(z, s_2)$ ← $G(s)$
               else if $G'(z, s_2) \not\sim G(s)$
                    then conflict ← true
         end;
      return($G'$)
end;
```

s is an input vector for the network; s_1 is the part of it received by the current node and s_2 is the part received by the remainder of the network. "$\not\sim$" denotes *incompatibility:*

$$a \not\sim b \Leftrightarrow a \neq \textit{don't care} \text{ and } b \neq \textit{don't care} \text{ and } a \neq b$$

The opposite is *compatibility:*

$$a \sim b \Leftrightarrow a = \textit{don't care} \text{ or } b = \textit{don't care} \text{ or } a = b$$

The procedure `reduce` can also be used if G itself contains *don't cares.* A *don't care* entry in G may be mapped into any entry of G' without causing a conflict.

4.4. GENERAL DECOMPOSITION ALGORITHM

This section describes an algorithm to solve the decomposition problem. Nothing is assumed about the structure of the given perceptron. Instead of designing a specific special-purpose algorithm that applies only to a restricted set of perceptrons, we develop a general-purpose strategy that applies to any perceptron. The algorithm is a search strategy consisting of the repeated application of a selection and reduction process. With a disjunctive binary tree the search strategy degenerates into a straightforward scheme.

The skeleton of the search strategy is presented first (section 4.4.1); the case of binary trees is discussed next (section 4.4.2); and the same strategy is then applied to general perceptrons (section 4.4.3).

4.4.1. Principles of the Search Algorithm

A brute-force solution to the decomposition problem might exhaustively generate all possible combinations of nodal assignments until a combination is found that produces the desired goal function. If redundant assignments are sought then the search continues until more solutions are found. In other words, the strategy consists of exhaustively generating all possible assignments α_j and for each assignment verifying whether $F_{\alpha_j} \equiv G$.

Denote N_G the number of ways G can be implemented on the given perceptron:

$$N_G = | \{ \alpha | F_\alpha \equiv G \} |$$

The probability that a randomly selected assignment α achieves the function

G is:

$$\text{Prob}[F_\alpha \equiv G] = \frac{N_G}{P}$$

where $P = p^l$ is the total number of possible assignments for the perceptron. Denote A the number of assignments examined before the correct one is found. Bounds for A are: $A_{\text{min}}=1$ and $A_{\text{max}}=P-N_G+1$. If the assignments are tried in a random order then A obeys a negative hypergeometric distribution with parameters N_G, $P-N_G$, and 1 [John77]. The mean of such a distribution is

$$A_{av} = \frac{P+1}{N_G+1}$$

The decomposition strategy discussed here generates a limited number of assignments (less than P) in an order such that A_{av} is as small as possible. The algorithm tries only assignments selected by a *selection criterion.* Its strength is measured by how well it succeeds in reducing A_{av}.

The selection criterion is global: which assignments it selects depends on the goal function and the entire structure of the perceptron. In our approach, however, the global assignment is decomposed into a sequence of nodal assignments, and the selection criterion is also local to each node. The nodes are treated in a sequential bottom-up fashion and at each node a few candidate functions are selected and tried one after the other. The decomposition algorithm then becomes a search through a tree structure. Every leaf of the tree corresponds to a complete assignment and A is the number of leaf nodes that are examined.

In a search procedure the computational time is not directly related to the number of leaf nodes examined, but depends on the size of that part of the tree that is traversed. We call S the number of nodes of the tree that are traversed. Bounds for S are: $S_{\text{min}}=l$, the number of nodes in the perceptron (the depth of the search tree), and $S_{\text{max}}=p^l=P$, the total number of assignments (the size of the exhaustive tree). There is no direct relationship between S_{av} and A_{av}.

The search procedure has 2 major components. First, a local *selection procedure* reduces the number of functions to be considered at each node from $p=|\phi|$ to some smaller number $p'=|\phi'|$. It is based on the observation that some functions can be excluded from examination and that a few functions are *likely* to achieve the given goal function. This component of the search procedure can be expressed as:

$$\phi' \leftarrow \mathrm{S}(\phi, G, i, \mathrm{P})$$

where S is the selection procedure, ϕ is the functional set of the node, G is the given goal function, i is the label of the current node, and P is the perceptron definition.

A second component of the search procedure is necessary because the function to be assigned to a node depends on previous assignments to other nodes. Therefore, the selection procedure S should be coordinated between different nodes. This coordination is accomplished by using a reduction computation between two nodes. The first node (in the bottom layer) selects an assignment based on the given goal function. It then passes a *residual* goal function, obtained by reduction, to the next node, which makes a selection based on this residual function, and so on. The coordination between nodes is therefore achieved by using successively modified goal functions.

The decomposition process is a repeated application of a selection step, followed by a local assignment and the corresponding reduction. This process is repeated for all the nodes of the network in a bottom-up order. Backtracking occurs when all selected local functions have been examined or when the reduction is impossible.

EXAMPLE 4.7: Figure 4.13 shows a simple perceptron. Suppose the given goal function is $G(x_1,x_2,x_3) = (x_1 \oplus x_2)(x_2+x_3)$. Treating the nodes in the order (2,3,1), the decomposition might proceed as follows:

$$\begin{aligned}
\text{Node 2:}\quad & G(x_1,x_2,x_3) = (x_1 \oplus x_2)(x_2+x_3) \\
& f_2(x_1,x_2) = x_1 \oplus x_2 \\
& G'(z_2,x_2,x_3) = z_2(x_2+x_3) \\
\text{Node 3:}\quad & G(z_2,x_2,x_3) = z_2(x_2+x_3) \\
& f_3(x_2,x_3) = x_2+x_3 \\
& G'(z_2,z_3) = z_2 z_3 \\
\text{Node 1:}\quad & G(z_2,z_3) = z_2 z_3 \\
& f_1(z_2,z_3) = z_2 z_3 \\
& G'(z) = z
\end{aligned}$$

This example demonstrates redundancy. Indeed, since

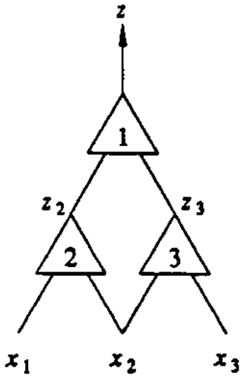


Figure 4.13: Example of a decomposition problem.

$(x_1 \oplus x_2)(x_2 + x_3) = \bar{x}_1 x_2 x_3$, an alternative solution might be:

Node 2: $G(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3$

$f_2(x_1, x_2) = \bar{x}_1 x_2$

$G'(z_2, x_2, x_3) = z_2 x_3$

Node 3: $G(z_2, x_2, x_3) = z_2 x_3$

$f_3(x_2, x_3) = x_3$

$G'(z_2, z_3) = z_2 z_3$

Node 1: $G(z_2, z_3) = z_2 z_3$

$f_1(z_2, z_3) = z_2 z_3$

$G'(z) = z$

Alternative assignments, obtained by equivalence or inversion operations, are listed below:

Node 2: $G(x_1,x_2,x_3)=\bar{x}_1x_2x_3$
$f_2(x_1,x_2)=x_1+\bar{x}_2$
$G'(z_2,x_2,x_3)=\bar{z}_2x_3$

Node 3: $G(z_2,x_2,x_3)=\bar{z}_2x_3$
$f_3(x_2,x_3)=\bar{x}_3$
$G'(z_2,z_3)=\bar{z}_2\bar{z}_3$

Node 1: $G(z_2,z_3)=\bar{z}_2\bar{z}_3$
$f_1(z_2,z_3)=\bar{z}_2\bar{z}_3$
$G'(z)=z$ □

4.4.2. Binary Tree Networks

We now apply the principles discussed in the previous sections to binary tree networks. Extensions to ternary or higher-order trees are trivial.

A *disjunctive* binary tree network is a network where the 2 input values of a node depend on a disjoint set of input variables, that is, each input variable is connected to only one node. In a *nondisjunctive* binary tree network the 2 inputs to a node may depend on an overlapping set of input variables. A nondisjunctive tree can be reduced to a disjunctive one by the introduction of virtual variables. Figure 4.14 shows a disjunctive binary tree with 8 inputs.

The number of virtual inputs for the network is $m=2^i$, where i is the number of layers in the tree. For a disjunctive tree $m=n$; for a

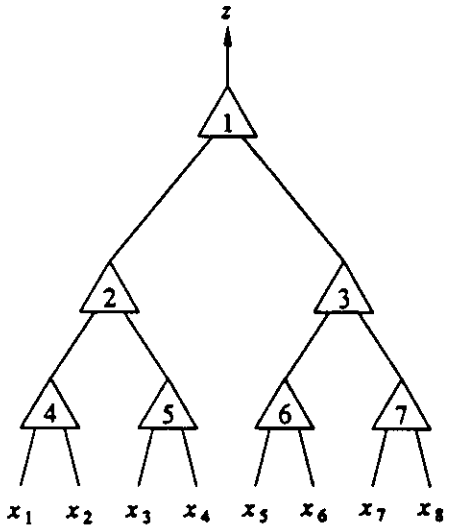


Figure 4.14: Example of a disjunctive binary tree network with 8 inputs. It has 7 nodes and 3 layers.

nondisjunctive tree $m > n$. The number of nodes in the network is $l = m-1$. A disjunctive tree is not complete ($Q < 2^{2^n}$; $\gamma<1$) and it exhibits only trivial redundancy. By contrast, a nondisjunctive tree can be complete provided there are enough fan-out connections in the inputs and its redundancy will typically be nontrivial.

The algorithm for decomposing a given Boolean function onto a binary tree (assuming a solution exists) works as follows. Starting with any node in the bottom layer, for instance node 4 in figure 4.14, a function is assigned using the decomposition chart with respect to $A_1 = \{x_1, x_2\}$ and $A_2 = \{x_3, \ldots, x_m\}$. (In a disjunctive tree, only a single function and its negation are possible. In a nondisjunctive tree, the assignment is not unique as a result of the *don't care* entries in the virtual decomposition chart, and a search is necessary.) Next, the residual function is determined by reduction. (It may generate *don't care* entries unless the tree is disjunctive.) The procedure treats the remaining nodes in a similar fashion, taking them one by one in a bottom-up order, that is, child nodes are assigned functions before their parent node. For the example of figure 4.14 two possible traversals of the nodes are (4,5,6,7,2,3,1) and (4,5,2,6,7,3,1).

To conclude, disjunctive binary trees have a simple decomposition strategy; no search is necessary and $S_{av}=S_{\min}=S_{\max}=l$. For a nondisjunctive binary tree, a search is typically required and $S_{av},S_{\max}>l$.

EXAMPLE 4.8: Figure 4.15 illustrates a decomposition in a nondisjunctive binary tree. The function for the network is $G(x_1,x_2,x_3)=x_1x_2+x_2x_3$. The figure shows the real and virtual decomposition charts. A possible assignment for node 2 would be $f_2(x_1,x_2)=\overline{x}_1x_2$, resulting in a residual function G' as shown. However, no assignment is possible for node 3 since its decomposition chart has 3 mutually incompatible columns, and backtracking is required. The assignment $f_2(x_1,x_2)=x_1x_2$ leads to a different residual function and $f_3(x_2^v,x_3)=x_2^vx_3$ is a possible assignment for node 3. This leads to a correct global assignment. □

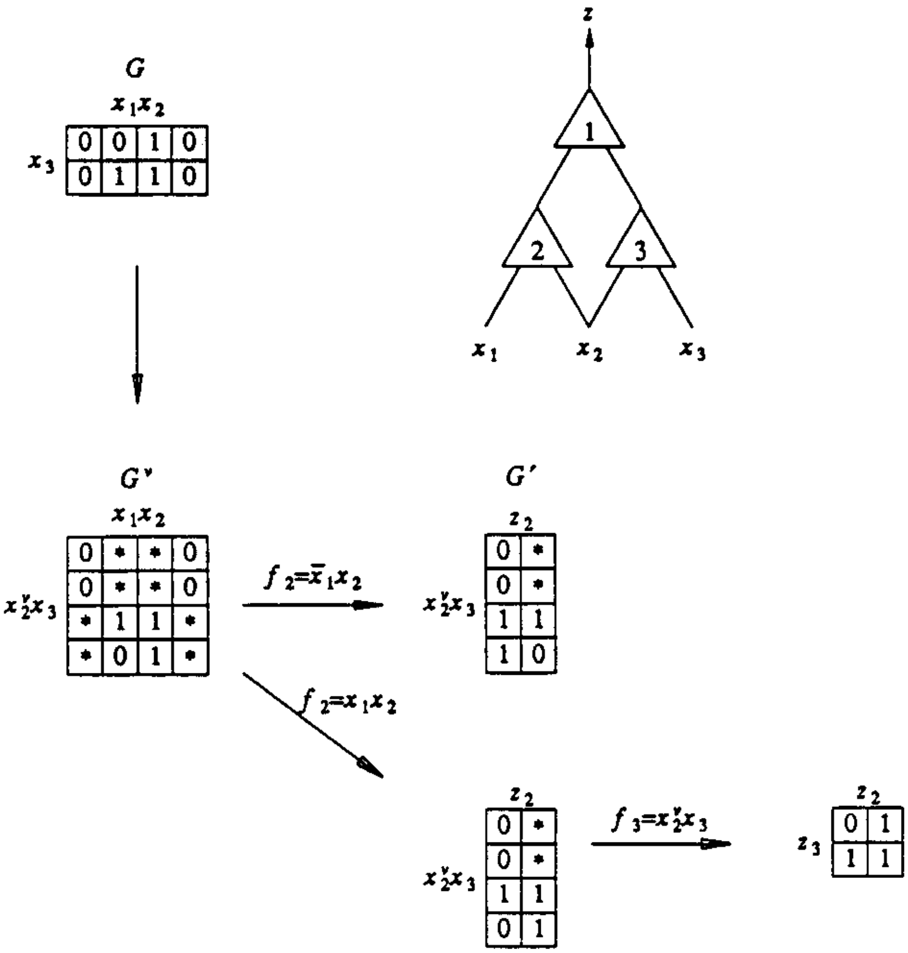


Figure 4.15: Example of backtracking in a nondisjunctive binary tree network.

4.4.3. General Perceptron Networks

The search algorithm is now extended to include all possible perceptrons. Two possible local selection criteria, resulting in two different algorithms, are presented. The *cautious* algorithm finds all possible (redundant) solutions, but it is by necessity defocussed and may require excessive search.

The *adventurous* algorithm is more focussed, but it does not aim at finding all the solutions. These two selection criteria are presented first. A preliminary analysis of the complexity of the adventurous algorithm follows. The last section presents a program that implements the adventurous algorithm.

4.4.3.1. Cautious selection criterion

Consider a node in the bottom layer of a perceptron and suppose it takes x_1 and x_2 as inputs. If neither x_1 nor x_2 are inputs to the remainder of the network then the decomposition is disjunctive and only two functional assignments (negations of each other) are applicable for that node. If the goal function contains *don't care* entries then multiple assignments may be possible and must be tried in sequence.

Now consider the case where x_2 is an input to the remainder of the network as well. The virtual decomposition chart must be used to determine the nodal assignment; mutually incompatible columns must have a different functional value. For node 2 in figure 4.16 only the columns G^{ν}_{01} and G^{ν}_{11} are mutually incompatible and should be assigned different functional values. Hence 8 nodal assignments are possible, as the figure shows. For this particular network the virtual decomposition chart has at most 2 pairs of mutually incompatible columns, namely $(G^{\nu}_{00},G^{\nu}_{10})$ and $(G^{\nu}_{01},G^{\nu}_{11})$. If these 2 pairs are indeed incompatible then only 4 local assignments must be tried. If only one pair is incompatible then 8 assignments are possible. In the extreme all 16 possible functions of 2 inputs have to be examined.

$$G \qquad \begin{array}{c|c|c|c|c} & \multicolumn{4}{c}{x_1x_2} \\ \hline x_3 & 0 & 0 & 1 & 0 \\ & 0 & 1 & 1 & 0 \end{array} \longrightarrow \qquad G^v \qquad \begin{array}{c|c|c|c|c} & \multicolumn{4}{c}{x_1x_2} \\ \hline & 0 & * & * & 0 \\ x_2^vx_3 & 0 & * & * & 0 \\ & * & 1 & 1 & * \\ & * & 0 & 1 & * \end{array}$$

POSSIBLE ASSIGNMENTS:

x_1x_2	0	0	1	0
x_1	0	0	1	1
$\bar{x}_1x_2$	0	1	0	0
$x_1 \oplus x_2$	0	1	0	1
$x_1 \otimes x_2$	1	0	1	0
$x_1+\bar{x}_2$	1	0	1	1
$\bar{x}_1$	1	1	0	0
$\bar{x}_1+\bar{x}_2$	1	1	0	1

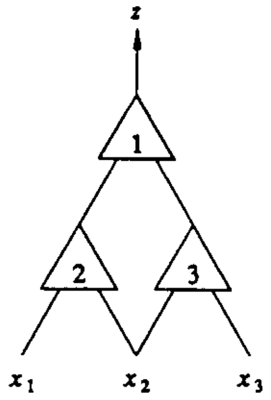


Figure 4.16: Example of cautious selection with one virtual variable x_2^v.

If *both* x_1 and x_2 are inputs to the remainder of the network as well then a pair of columns is never incompatible (see figure 4.17) and therefore all 16 local functions have to be examined.

In general, if a node has k inputs, v of which are inputs to the rest of the network as well ($0 \le v \le k$), and if the goal function is completely specified, then the number of functional options for that node is *at least* 2^{2^v}. The selection procedure reduces ϕ to ϕ' with the size of ϕ' bounded by

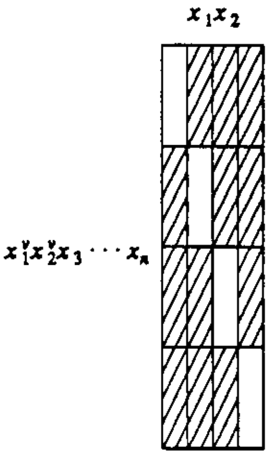


Figure 4.17: Virtual decomposition chart with two virtual variables x_1^v and x_2^v.

$$2^{2^v} \leq p' = |\phi'| \leq 2^{2^k}$$

If the given goal function contains *don't cares* then the lower bound is larger. Hence, *don't cares* increase the amount of search needed to find a decomposition. Fan-out connections in the network introduce *don't care* entries in the residual goal truth tables, and they constitute a major source of search complexity in the decomposition problem.

The cautious selection procedure S_c, selecting functions that have different values for mutually incompatible columns of the virtual decomposition chart, is listed below:

```
function $S_c(\phi, G, i, P)$;
begin $G^v \leftarrow$ virtual($G$,i,P);
      $\phi' \leftarrow \emptyset$;
      for all $f \in \phi$
      do begin retain $\leftarrow$ true;
               for all $(s,s') \in (\{0,1\}^k)^2$
               do if $(G_s^v \neq G_{s'}^v)$ and $(f_s = f_{s'})$
                  then retain $\leftarrow$ false;
               if retain
               then $\phi' \leftarrow \phi' \cup f$
         end;
      return($\phi'$)
end;
```

virtual is a function that generates the virtual truth table. G_s^v is the s-th column of the virtual decomposition chart; f_s is the s-th entry of the local function.

The program listed above implements the test

$$\forall\, s,s' \in \{0,1\}^k : G_s^v \neq G_{s'}^v \Rightarrow f_s \neq f_{s'}$$

This test, however, is automatically included as part of the reduction process. Indeed, the reduction will return a conflict if and only if two mutually incompatible columns of the virtual decomposition chart receive the same functional value. In other words, a function selected by the cautious selection will not cause a conflict in the reduction step, and, vice versa, a conflict in the reduction computation occurs only for a function not selected by the cautious criterion. Hence, S_C is redundant and one may as well use the reduction process to make the selection. Therefore, the procedure can be simplified to:

```
function $S_c(\phi, G, i, P)$;
begin return($\phi$)
end;
```

In other words, the cautious criterion does no selection but feeds the reduction phase with *all possible* local functions. This program is therefore doing the maximum amount of search and relies only on the reduction step to cut down the number of assignments. In return, it generates all possible redundant assignments after successive backtrackings.

4.4.3.2. Adventurous selection criterion

We now discuss a possibility to further reduce the size of ϕ', based on the real decomposition chart. This decomposition chart generally contains more than two mutually incompatible groups of columns. How can local functions be selected based on this contradictory decomposition chart?

EXAMPLE 4.9: In figure 4.18 there are 3 mutually incompatible classes of columns in the real decomposition chart of node 2, namely $C_1=\{G_{00},G_{11}\}$, $C_2=\{G_{01}\}$, and $C_3=\{G_{10}\}$. We select three functions (plus their negations) by mapping these 3 mutually incompatible classes into 2 groups and assigning the same functional value to columns of the same group. The conflict that results from assigning two incompatible columns the same value is ignored for the time being; we assume it will be taken care of by the remainder of the network. Three groupings are possible: $\{C_1 \cup C_2, C_3\}$, $\{C_3 \cup C_1, C_2\}$, and $\{C_2 \cup C_3, C_1\}$. Each grouping results in a functional assignment, as figure 4.18 shows. □

G

	x_1x_2			
x_3	0	0	1	0
	0	1	1	0

POSSIBLE ASSIGNMENTS:

x_1x_2	0	0	1	0
$\bar{x}_1x_2$	0	1	0	0
x_2	0	1	1	0
$\bar{x}_2$	1	0	0	1
$x_1+\bar{x}_2$	1	0	1	1
$\bar{x}_1+\bar{x}_2$	1	1	0	1

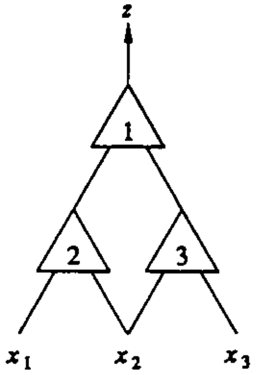


Figure 4.18: Example of adventurous selection.

This selection is based on the incompatibilities in the real decomposition chart only and does not need any knowledge of how many or which local variables are inputs to the remainder of the network as well.

If the goal function G contains *don't care* entries then one column (containing *don't care* entries) could be compatible with two other columns that are mutually incompatible (see for example figure 4.19). Two alternative assignments are then possible, and both must be tried.

EXAMPLE 4.10: Figure 4.20 illustrates that the adventurous algorithm does not necessarily generate *all* solutions. With the goal function $G(x_1,x_2,x_3)=x_1x_2$, the only functions tried for node 2 are $f_2(x_1,x_2)=x_1x_2$

x_3 \ x_1x_2	00	01	11	10
0	1	1	1	0
1	*	0	1	0

Figure 4.19: Column G_{00} is compatible with both G_{01} and G_{11}, but G_{01} and G_{11} are mutually incompatible.

and its negation. However, the choice $f_2(x_1,x_2)=x_1$ and $f_3(x_2,x_3)=x_2$, also a valid solution, is not found. □

G

x_3 \ x_1x_2				
	0	0	1	0
	0	0	1	0

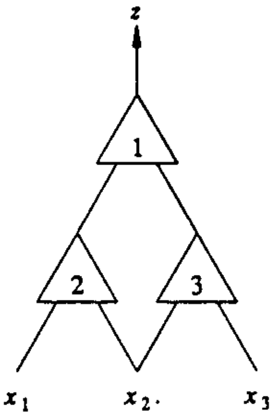


Figure 4.20: Example showing that the adventurous algorithm does not necessarily generate all solutions to a decomposition problem.

The adventurous algorithm neglects some correct assignments but reduces the total size of the search tree. Is it possible that the algorithm excludes *all* solutions from consideration? In other words, is it possible for the algorithm to be too adventurous and work itself into a corner, never finding a correct assignment even though one exists?

The adventurous criterion is intuitively appealing. Consider the example of figure 4.20: why should $f_2(x_1,x_2)=x_1$ or $f_2(x_1,x_2)=\overline{x}_1x_2$ be tried? They do not cause a reduction conflict and might be part of a solution, but $f_2(x_1,x_2)=x_1x_2$ is definitely the most appropriate choice, the most likely to lead to a correct assignment. In general, different local assignments lead to different residual functions G' and it depends on the structure of the remainder of the network which G' is implementable.

The program for the adventurous algorithm is listed below:

```
function $S_a(\phi, G, i, P)$;
begin $\phi' \leftarrow \emptyset$;
        for all $f \in \phi$
        do begin retain ← true;
                        for all $(s, s') \in (\{0,1\}^k)^2$
                        do if $(G_s \sim G_{s'})$ and $(f_s \neq f_{s'})$
                           then retain ← false;
                        if retain
                        then $\phi' \leftarrow \phi' \cup f$
            end;
        return($\phi'$)
end;
```

In summary, the adventurous selection criterion is the complement of the cautious criterion. Instead of assigning *different* functional values to *incompatible* columns of the *virtual* decomposition chart, it assigns *equal* functional values to *compatible* columns of the *real* decomposition chart. The relationship between both selection criteria is as follows. The cautious selection is based on the formal property (theorem 4.5)

$$\forall\ s,s' \in \{0,1\}^k : G_s^v \nsim G_{s'}^v \Rightarrow f_s \neq f_{s'} \tag{4.1}$$

The adventurous selection uses:

$$\forall\ s,s' \in \{0,1\}^k : G_s \sim G_{s'} \Rightarrow f_s = f_{s'} \tag{4.2}$$

The latter is *not* a theorem (we showed a counterexample above) but a heuristic. By contrast, (4.1) is a theorem that holds *for all* goal functions and *for all* assignments implementing the given function. It remains to be proven that *for all* goal functions *there exists* a corresponding assignment for which (4.2) holds. This question is left for future research.

4.4.3.3. Characteristics of the adventurous algorithm

The combined adventurous decomposition program is listed below:

```
procedure decomp($G$, $i$, $\alpha$, P);
begin if $i$=0
      then print assignment $\alpha$
      else begin $\phi' \leftarrow S_a(\phi, G, i, P)$;
                 for all $f_{i,j} \in \phi'$
                 do begin $\alpha' \leftarrow \alpha \cup \{i \rightarrow f_{i,j}\}$;
                          $G' \leftarrow$ reduce($G$, $f_{i,j}$, conflict);
                          if not conflict
                          then begin $i' \leftarrow$ next($i$, P);
                                     decomp($G'$, $i'$, $\alpha'$, P)
                               end
                    end
           end
end;
```

G and G' are respectively the goal function and the residual goal function for node i; the latter is passed to the next node. The procedure `next` calculates the label of the next node in a bottom-up fashion; label 0 is returned when the top node is reached. If i=0 then the top node has been assigned a local function and, since the corresponding reduction succeeded, the complete assignment implements either the given goal function or its negation.

The performance of the decomposition algorithm can be measured by the amount of memory required per node and the number of search steps before a solution is found. The amount of memory required for each node is equal in size to the goal truth table. However, the goal truth table decreases in size as the algorithm proceeds through the network. For example, the sizes of the goal truth tables for two different node traversals of the network of figure 4.21 is shown in table 4.2. The first traversal is superior in terms of memory requirements.

Assume each node has k inputs and the network has n inputs. The number of different columns in the real decomposition chart is less than 2^k, the total number of columns in the decomposition chart. Additionally, the length of each column is 2^{n-k} entries and therefore at most $2^{2^{n-k}}$ columns are possible. If C represents the number of different columns in the decomposition chart then

$$C \leq \min\{2^k, 2^{2^{n-k}}\} = C_{\max}$$

This upper bound is smallest for $k=2$ and for $k=n-1$ ($C_{\max}=4$) and is largest for $k=2^{n-k}$, that is, $\log_2(k)+k=n$. If C different columns exist then the adventurous criterion will select 2^C local functions, which is bounded by $2^{C_{\max}}$.

The size of the goal function changes from node to node, and so does $C_{\max}$. If n_i is the number of network inputs when node i is considered then

$$S_{\max} \leq U = \prod_{i=1}^{l} 2^{C_i} \quad \text{with} \quad C_i=\min\{2^k, 2^{2^{n_i-k}}\}$$

A tighter upper bound for $S_{\max}$ would have to take into account the assignments excluded by the reduction conflict detection. Furthermore, this would still only give an upper bound, namely the number of steps if the resulting tree is traversed entirely.

Simulation showed for the 3-input perceptron of figure 4.22: $S_{\max}=99$ and $S_{av}=26$. This is certainly an enormous improvement over the upper limit $U=2^{22}$. Further experimental results, using the program described in the next section, are given in table 4.3. Different numbers in the column S are

node	7	8	9	10	4	5	6	2	3	1	total
size	32	32	32	32	16	16	16	8	8	4	196
node	7	8	4	9	5	2	10	6	3	1	total
size	32	32	32	32	32	32	32	16	8	4	252

Table 4.2: Size of the memory required at each node for different traversals of the nodes of the perceptron shown in figure 4.21.

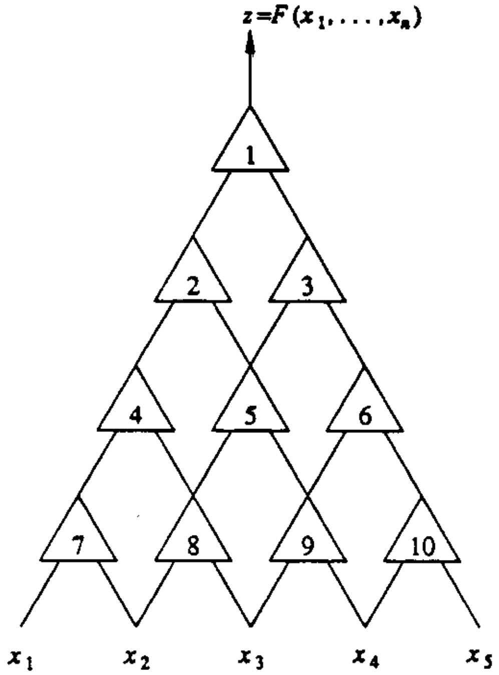


Figure 4.21: A simple 5-input perceptron.

The number of search steps necessary, S, depends not only on the particular perceptron structure, but also on the given goal function. The following analysis leads to an upper bound for S.

for different goal functions G.

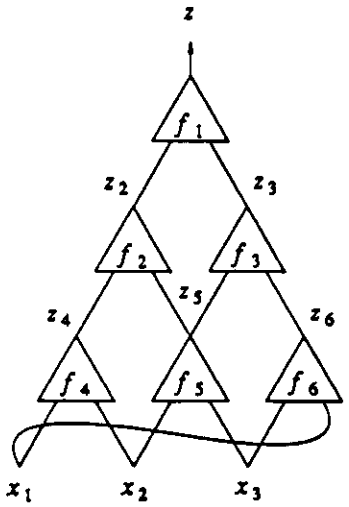


Figure 4.22: Example of a 3-input perceptron.

n	k	l	S	U
3	2	6	11, 55, 58	2^{22}
4	3	4	95, 246, 288	2^{22}
4	2	12	158, 613, 775	2^{32}
4	2	24	61, 178, 205	2^{44}
4	2	28	236, 592, 695	2^{110}
7	2	18	39	2^{40}

Table 4.3: Experimental data for the adventurous decomposition program.

4.4.3.4. Implementation

The adventurous algorithm decomp has been programmed in Pascal. It uses a more efficient formulation of the adventurous selection process. Instead of *testing* all members of ϕ with the criterion, it *generates* the set ψ of all functions that satisfy the criterion. It then checks each one for membership in ϕ. In other words, S_a is implemented as:

```
function $S_a(\phi, G, i, P)$;
begin $\psi \leftarrow$ generate-all($G$,i,P);
      $\phi' \leftarrow \psi \cap \phi$;
      return($\phi'$)
end;
```

If ϕ is complete then $\phi' \equiv \psi$. This approach is more efficient in time and in storage requirements than the version presented earlier.

The program was written to be able to invoke itself recursively on a subnetwork. This allows the user to define a perceptron as a network of atomic nodes and then use this perceptron as a *macro-node* or module in the definition of a larger perceptron, and so on (see for example figure 4.23). The program decomposes the goal function and assigns functions to each subnetwork assuming they are atomic nodes of the network. It then further subdecomposes the assigned functions for these smaller networks, and so on. The decomposition is organized in a depth-first order, that is, the subdecompositions are treated before the next macro-node is considered. The decomposition backtracks if the current subnetwork cannot implement its assigned function.

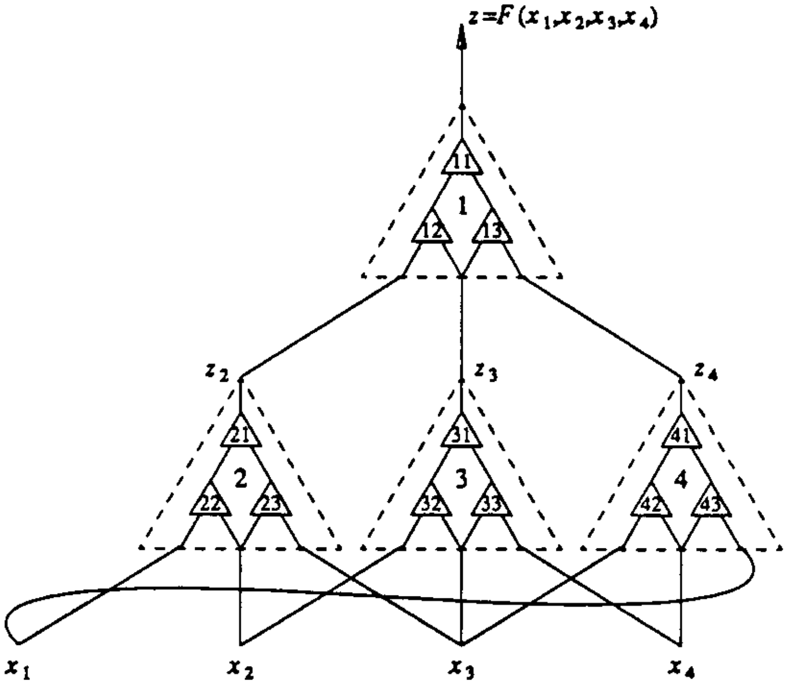


Figure 4.23: Example of a multilevel perceptron.

As a result of this multilevel strategy, backtracking can be done in a more radical fashion than would be possible in a single-level definition of the same perceptron. The rationale is the following. If a dead-end is found for example at macro-node 3 of the perceptron in figure 4.23 (that is, all the options selected for this node generate a reduction conflict) then there is no need to continue with different assignments for the nodes in the macro-node 2. In other words, there is no need to backtrack inside the subnetwork 2 and try different assignments for the nodes 21, 22, or 23 with the same assignment for the subnetwork 2. Such different assignments would only cause the same dead-end at macro-node 3. Hence, the backtracking can be continued until the next assignment for macro-node 2 is found.

The program reads the definition of the perceptron P=[N,*z*,**x**,I] from a file. N defines the node, *z* and **x** label the network output and inputs, and I defines the interconnection structure. The perceptron specifications are written in a simple language whose context-free grammar is listed below.

```
<networkdescription> ::= <atomdefinition> ; <networkdefs> .
<atomdefinition> ::= define <representation>
                        functions { <truthtableset> } |
                     define <representation>
                        functions complete
<representation> ::= <identifier> =
                        <identifier> ( <parameterlist> )
<identifier> ::= <letterordigit> [ <identifier> ]
<letterordigit> ::= <letter> | <digit>
<parameterlist> ::= <identifier> [ , <parameterlist> ]
<truthtableset> ::= <truthtable> [ , <truthtableset> ]
<truthtable> ::= <zeroorone> [ <truthtable> ]
<networkdefs> ::= <networkdefinition> [ ; <networkdefs> ]
<networkdefinition> ::= define <representation>
                        interconnect { <nodelist> }
<nodelist> ::= <nodedefinition> [ ; <nodelist> ]
<nodedefinition> ::= <identifier> : <representation>
```

The definition of the perceptron of figure 4.22 is listed below.

```
% Definition of a 3-input perceptron with 6 nodes.

% Definition of the node
define z = atom(x1,x2) functions complete;
% Definition of the 3-input perceptron
define z = F(xI,x2,x3)
       interconnect { node4 : z4 = atom (x1,x2);
                      node5 : z5 = atom (x2,x3);
                      node6 : z6 = atom (x3,x1);
                      node2 : z2 = atom (z4,z5);
                      node3 : z3 = atom (z5,z6);
                      node1 : z  = atom (z2,z3) }.
```

The definition of the perceptron of figure 4.23 is:

```
% 4-input perceptron built with 3-input perceptrons.

% Definition of the atomic node
define y = atom(x1,x2) functions complete;
% Definition of the 3-input perceptron:
define z = f(x1,x2,x3)
       interconnect { a2 : z2 = atom(x1,x2);
                      a3 : z3 = atom(x2,x3);
                      a1 : z  = atom(z2,z3) };
% Definition of the 4-input perceptron:
define z = F(x1,x2,x3,x4)
       interconnect { m2 : z2 = f(x1,x2,x3);
                      m3 : z3 = f(x2,x3,x4);
                      m4 : z4 = f(x3,x4,x1);
                      m1 : z  = f(z2,z3,z4) }.
```

Lines starting with a "%" are comment lines. The first few lines of the definition specify $N\equiv(k,\phi)$. Each **define** statement lists *z* and **x** and the following **interconnect** statement defines I.

The basic node (called atom in both cases) could be defined arbitrarily by giving a list of local truth tables. For example:

```
% 2-input node implementing AND, LEFT, RIGHT, OR
define z = atom(x1,x2)
       functions { 0001, 0011, 0101, 0111 } ;
```

The decomposition program treats the nodes (or macro-nodes) in the order given by the perceptron specification. Therefore the node connections

I should be ordered in a bottom-up sequence. A label cannot be used as input to a node unless it was defined earlier as the output label of another node or unless it is a network input. Additionally, the subnetworks must be defined in a depth-first order, meaning that the node must be defined first, then the subnetworks, and finally the global network. A subnetwork must be defined before it can be used as macro-node in another interconnection definition.

This language allows the specification of any perceptron with only a single node, as defined in section 4.1. The adventurous algorithm described earlier, however, does not require all nodes to be identical, and the program and definition language could be extended in this direction. Nevertheless, except for this restriction the program is completely general. It does not make any assumptions regarding the completeness of the node, the completeness of the network, the structure of the interconnections, or the size of the nodes. Furthermore, it finds multiple solutions, but not all.

Below is a partial script of the program using the 3-input perceptron of figure 4.22. The goal function is $G(x_1,x_2,x_3) = x_1x_2+x_2x_3+x_3x_1$. Inputs typed by the user are in **bold** face.

```
specify network file: Nw3.6
specify truth table of 3 variables (x1,x2,x3):
01101011
   1: 1 | node4: 0000... ok.
   2: 1 | node5: 0000... conflict.
   3: 1 | node5: 0010... conflict.
   4: 1 | node5: 0100... conflict.
   5: 1 | node5: 0110... ok.
```

```
     6: 1 | node6: 0000... conflict.
     7: 1 | node6: 0001... conflict.
     8: 1 | node6: 0100... conflict.
     9: 1 | node6: 0101... conflict.
    10: 1 | node6: 1010... conflict.
    11: 1 | node6: 1011... conflict.
    12: 1 | node6: 1110... conflict.
    13: 1 | node6: 1111... conflict.
    14: 1 | node5: 1001... ok.
              .
              .
              .
    45: 1 | node3: 0101... conflict.
    46: 1 | node3: 0110... ok.
    47: 1 | node1: 0000... conflict.
    48: 1 | node1: 0111... ok.
  implementation found after 48 steps:
  1 | node4: 0001
  1 | node5: 0011
  1 | node6: 0110
  1 | node2: 0001
  1 | node3: 0110
  1 | node1: 0111
  continue? [y,n,#] no
  search aborted.

  number of steps:    48
  maximum stack size: 366 words
  other memory used:  286 records

  do you want to run another decomposition? no
```

An example of a decomposition problem for the perceptron of figure 4.23 with goal function $G(x_1,x_2,x_3,x_4)=x_1x_3+x_2(x_1\oplus x_4)$ is given below:

```
  specify network file: Nw4.4.3
  specify truth table of 4 variables (x1,x2,x3,x4):
  0110100100110011
     1: 1 | m2: 00000000... ok.
     2: 2 | m2.a2: 0000... ok.
  3: 2 | m2.a3: 0000... ok.
  4: 2 | m2.a1: 0000... ok.
  5: 1 | m3: 00000000... conflict.
  6: 1 | m3: 00010010... conflict.
  7: 1 | m3: 00100001... conflict.
  8: 1 | m3: 00110011... conflict.
  9: 1 | m3: 01001000... conflict.
 10: 1 | m3: 01011010... ok.
 11: 2 | m3.a2: 0000... conflict.
 12: 2 | m3.a2: 0011... ok.
 13: 2 | m3.a3: 0000... conflict.
 14: 2 | m3.a3: 0101... ok.
 15: 2 | m3.a1: 0000... conflict.
 16: 2 | m3.a1: 0110... ok.
 17: 1 | m4: 00000000... conflict.
 18: 1 | m4: 00000101... conflict.
 19: 1 | m4: 00001010... conflict.
 20: 1 | m4: 00001111... conflict.
 21: 1 | m4: 01010000... conflict.
 22: 1 | m4: 01010101... conflict.
 23: 1 | m4: 01011010... conflict.
 24: 1 | m4: 01011111... conflict.
 25: 1 | m4: 10100000... conflict.
 26: 1 | m4: 10100101... conflict.
 27: 1 | m4: 10101010... conflict.
 28: 1 | m4: 10101111... conflict.
 29: 1 | m4: 11110000... conflict.
 30: 1 | m4: 11110101... conflict.
 31: 1 | m4: 11111010... conflict.
 32: 1 | m4: 11111111... conflict.
 33: 1 | m3: 01101001... ok.
         .
         .
         .
145: 2 | m4.a1: 0100... ok.
146: 1 | m1: 00000000... conflict.
147: 1 | m1: 00000101... conflict.
148: 1 | m1: 00101010... ok.
149: 2 | m1.a2: 0000... conflict.
150: 2 | m1.a2: 0111... ok.
151: 2 | m1.a3: 0000... conflict.
152: 2 | m1.a3: 0001... conflict.
```

```
 153: 2 | m1.a3: 0010... conflict.
 154: 2 | m1.a3: 0011... conflict.
 155: 2 | m1.a3: 0100... conflict.
 156: 2 | m1.a3: 0101... ok.
 157: 2 | m1.a1: 0000... conflict.
 158: 2 | m1.a1: 0010... ok.
implementation found after 158 steps:
1 | m2: 00000101
2 | m2.a2: 0011
2 | m2.a3: 0101
2 | m2.a1: 0001
1 | m3: 01101001
2 | m3.a2: 0110
2 | m3.a3: 0101
2 | m3.a1: 0110
1 | m4: 01010000
2 | m4.a2: 0011
2 | m4.a3: 0101
2 | m4.a1: 0100
1 | m1: 00101010
2 | m1.a2: 0111
2 | m1.a3: 0101
2 | m1.a1: 0010
continue? [y,n,#] no
search aborted.

number of steps:    158
maximum stack size: 802 words
other memory used:  453 records

do you want to run another decomposition? no
```

Answering the question "`continue? [y,n,#]`" with "**`yes`**" will make the program backtrack and find another solution.

4.5. CONCLUSIONS

The decomposition problem in its most general form is a hard problem that can require a large amount of search. Sometimes a solution is found early in the search, but there is no guarantee that this will always be the case, or that it will be true on the average. If on the average the size of ϕ is reduced to $p'=|\phi'|$ then it would require $O((p')^l)$ steps to traverse the tree in its entirety. In the worst case, the solution for the decomposition problem requires an almost exhaustive search.

The main issue is not so much the amount of search needed, but the nature of the *responsibility assignment* task. When can an assignment of functional responsibility be done uniquely? When is it simple and when is it difficult?

The multilayered nature of the perceptron *by itself* is not an obstacle in determining which part of the network should implement a certain piece of the goal function. The example of the disjunctive tree shows that for any goal function implementable on this network the assignments for all the nodes can always be decided by a straightforward algorithm. The number of node assignments is linear in the number of nodes in the network. What is it that makes the assignment so simple in a disjunctive binary tree and so complex in other networks?

The size of the node versus the size of the whole network is an important issue. If the node or subnetwork to be assigned responsibility is only a little smaller in size than the whole network, then the problem is simpler than if the node is much smaller. The goal function must then satisfy strict requirements for the assignment to be focussed to one or a few possibilities (as in a disjunctive binary tree). In the former case most goal functions allow a straightforward solution. If the goal happens to depend only on the variables that are inputs to a given node then the assignment is solved trivially.

In general, the sources of difficulty are the fan-out connections (external or internal). Fan-outs introduce virtual variables and hence *don't cares* in the virtual decomposition chart, or an opportunity for many incompatible columns in the real decomposition chart. As a result, the local selection (both cautious and adventurous) is not focussed and the decomposition requires a search. Furthermore, in such a case the responsibility assignment is not unique, because the structure is redundant.

The fundamental reason behind the necessity for a search is that the assignment cannot be entirely local and must use some global knowledge of the network structure. If multiple assignments are possible at a given node, each one will produce a different residual goal. Deciding which local assignments are correct requires knowing which residual goals are implementable on the remaining network. Coordinating local assignments with global requirements is therefore of crucial importance. The local assignments depend not only on the assignments previously made to other nodes, but also on the structure of the remainder of the network (the part that has not yet been considered by the search). The question whether a (residual) goal is implementable on a (remainder of a) perceptron is different from the problem of finding an implementation. The former is a binary question; the latter is a search for a particular solution. We do not yet know whether the former is computationally simpler.

Different traversals of the nodes are possible, opening a possible improvement to the decomposition algorithm. Some orderings might lead to a faster solution (less backtracking) than others. Which order should be preferred? It depends on the goal function. The node with the simplest assignment, the smallest set of selected functions ϕ', should be taken first. This node has the smallest number of mutually incompatible groups of columns in its decomposition chart, or the maximum number of mutually compatible columns. Finding this node is different from the original decomposition problem, and its complexity remains to be evaluated. An ordering of nodes such that at each step $p'=|\phi'|$ is minimal could reduce the complexity of the problem substantially. Finding this optimal order of traversal can be facilitated by using parallelism. Assume that all the nodes in the bottom layer receive the global goal at the same time and compute ϕ' concurrently. A global control mechanism can then designate a single winner to execute the reduction step. The same process can be repeated for the nodes in the bottom layer of the remainder of the network, and so on. By using parallelism the nodes are treated in an *optimal* order at no cost in time. Comparing the size of this optimal search path with the size of an average search path would determine if the extra cost of parallelism is offset by the improvements in speed.

CHAPTER 5

TOWARDS LEARNING BY EXAMPLE IN PERCEPTRONS

There exists a wide spectrum of possible schemes for using perceptrons. Decomposition is at one extreme end of this spectrum. By contrast, the current chapter investigates opportunities for learning by example in perceptrons by relaxing the requirement that the entire goal function be specified all at once. We do not treat or define learning in general, but concentrate exclusively on a specific form of learning by example in perceptrons. The work by Rosenblatt dealt only with this form of learning, and the later research (see section 5.2) has followed the same learning paradigm.

Recently, the field of machine learning has become a very active and much broader area of research [Mich83, Mich86]. However, almost all the current work concentrates on the implementation of learning on a sequential von Neumann computer. Little of the modern literature on the subject of learning deals with the direct implementation of learning paradigms on perceptrons, and then only limited classes of perceptrons (see sections 5.2 and 5.3). This chapter presents an overview of this work and develops a possible improvement to an existing scheme for learning by example in disjunctive binary trees. Decomposition charts are again the basic tool in this study.

Section 5.1 gives an overview of the concepts of learning and learning by example, specifically in the context of perceptrons. Section 5.2 reviews the existing literature on learning by example in threshold gate perceptrons. Next, section 5.3 discusses two strategies for learning by example in digital perceptrons and presents an extension of this work. Some conclusions are presented in section 5.4.

5.1. OVERVIEW OF LEARNING BY EXAMPLE IN PERCEPTRONS

This section first presents an introductory overview of machine learning and learning by example. A formal definition and detailed discussion of learning by example in perceptrons is given next. Two different implementations, perceptrons with central or distributed control, are discussed and possible schemes for learning by example in these implementations are presented.

5.1.1. Introduction

A precise definition of the term *learning* is virtually impossible; no commonly-accepted definition is available in the literature. Intuitively, learning implies improving performance over time, or acquiring new skills or knowledge. Learning also implies the ability to absorb unspecific or incomplete "instructions" and change internal operations to satisfy these external requirements. The latter are presented in an incremental fashion and hence the instructions become more complete and more specific over time. In summary, the learning system must be able to internalize this unspecific information and integrate it with what was previously learned. The distinction between *programming* and *learning,* however, is a matter of degree. Programming is one extreme embodiment of a broad spectrum of learning schemes.

In a learning system there is a clear separation between the *learner* and its environment; some knowledge is transferred *from* the environment *to* the learner. The *teacher* is part of the environment and may be a human or another machine. In *supervised* learning a separate teacher is involved in the learning process; in *unsupervised* learning the system learns by itself without explicit help from a teacher.

In learning by example the learner must optimize its operation based on a set of examples. We assume here that the examples are provided by the teacher and that each example is accompanied by an immediate signal specifying the desired system output. Learning by example as discussed in this chapter is therefore supervised learning as it depends on explicit guidance from a teacher in its learning phase.

There exist two different schemes for learning by example. In one scheme each example is presented to the learner accompanied by the corresponding desired output. In another scheme the learner classifies each example and the teacher then specifies whether the output was correct. In the former scheme the teacher's input is a *feedforward* signal; in the latter scheme it is a *feedback* signal. The learner is expected to change its operation in response to this signal.

Two properties are important in a strategy for learning by example. First, it should correctly classify the examples that are shown in the learning phase. Secondly, new examples should be assigned to the *most likely* class based on the earlier examples. The latter property is called *generalization.* It assumes a metric in the input space to decide the nearest example previously shown. This metric depends on the application domain, the nature of the examples, the type and form of the knowledge to be learned, and many other issues that differ from case to case.

5.1.2. Learning by Example in Perceptrons

A perceptron is a network of combinational nodes and implements a Boolean function. This Boolean function constitutes the *knowledge* to be learned by the perceptron. Initially the system implements an arbitrary function, and the purpose of the examples is to convey a desired Boolean function to the perceptron.

An *example* is a completely specified input vector for the perceptron. All the input variables of an example have a specific value. In other words, an example corresponds to a *minterm.* Although some related work allows *don't cares* in the examples [Vali84, Mart86], virtually all extant work in perceptrons uses completely specified examples.

In a session of learning by example, the teacher presents a discrete *sequence of examples* to the inputs of the perceptron. This sequence is represented as

$$\mathsf{E} = [\mathbf{x}_1, \mathbf{x}_2, \ldots] = [\mathbf{x}_i]$$

In a theoretical treatment E can be infinitely long, but in a practical implementation E is always finite. In what follows we will assume the sequence of examples to be of finite length L.

The examples in the sequence are not necessarily all different; typically, some examples reappear. Each subsequence of examples $\mathsf{E}_k = [\mathbf{x}_1, \ldots, \mathbf{x}_k]$ has a corresponding set of distinct examples:

$$\mathsf{D}_k = \{\mathbf{x}_i \mid i \leq k \text{ and } \mathbf{x}_i \neq \mathbf{x}_j, \forall j < i\}$$

obtained by deleting all duplicate examples from E_k.

Each example presented to the perceptron is accompanied by a specification of the corresponding value for the goal function. In other words, when an example $\mathbf{x}_i$ is presented, the teacher determines the correct output y_i and sends this value to the perceptron. An example that reappears in the sequence must have identical output values.

We view this sequence of events as an incremental specification, one minterm at a time, of a goal function G for which $\forall i : G(\mathbf{x}_i) = y_i$. As long as the examples have not been shown exhaustively, the goal function is specified only partially. We define the *partial* goal function after k examples, G_k, to be the function that contains *don't care* values for examples not in the sequence E_k. In other words

$$G_k(\mathbf{x}_i) = \begin{cases} y_i = G(\mathbf{x}_i) & \text{if } \mathbf{x}_i \in \mathrm{D}_k \\ \textit{don't care} & \text{if } \mathbf{x}_i \notin \mathrm{D}_k \end{cases}$$

If *all* 2^n possible examples are in the sequence then $G_L \equiv G$, which is a completely specified function.

A network function F is *compatible* with the partial goal function G_k, represented as $F \sim G_k$, if G_k can be made identical to F by filling in the *don't care* values of G_k. In other words

$$F \sim G_k \iff \forall\ \mathbf{x}_i \in \mathrm{D}_k : F(\mathbf{x}_i) = G_k(\mathbf{x}_i) = y_i$$

We view the sequence of examples and teaching signals as a sequence of partially specified goal functions $[G_i]$. The task of the control of the perceptron is to derive a sequence of assignments $[\alpha_i]$, and hence a sequence of network functions $[F_{\alpha_i}]$, from this sequence $[G_i]$. This incremental transfer of a goal function G is one form of learning by example and will be studied in the remainder of this chapter.

A strong requirement for a learning algorithm, called *strict convergence*, is the following:

$$\forall i : F_{\alpha_i} \sim G_i$$

If a learning system exhibits strict convergence then each example needs to be presented only once. This requirement can be relaxed to

$$F_{\alpha_L} \sim G_L \tag{5.1}$$

This means that the network function F *converges* to G_L and that it is compatible with G_L when enough examples have been shown.

Note in (5.1) the appearance of G_L instead of the real goal function G. G_L is identical to G only if D_L is the complete set $\{0,1\}^n$. Hence (5.1) does not specify the values of F_{α_L} for input vectors not in D_L. No immediate information is available regarding these examples, but it is possible to extrapolate, that is, to generalize. For instance, F could assign an unseen example the same value as the closest example in D_L. Such an approach is natural in an analog domain with continuous distances. In a general digital domain of Boolean logic, however, generalization assumes a priori knowledge about the class of goal functions to be learned. If such knowledge is not available then all goal functions are possible. If *all* possible examples are shown then D_L is complete ($G_L \equiv G$) and no generalization is needed.

5.1.3. Perceptrons with Centralized Control

How can the incremental transfer of G proceed in a perceptron with a central controller? In particular, how can the desired output value $y_i = G(\mathbf{x}_i)$ be communicated to the controller of the perceptron? Two approaches are possible.

In one approach the teacher specifies the desired goal value for an example by sending a positive or negative *feedback* to the controller [Rose62b]. This feedback signal specifies whether the perceptron made the correct decision for the current example. If the feedback is positive then there is no reason to change the assignment, hence $\alpha_i=\alpha_{i-1}$ and $F_{\alpha_i}=F_{\alpha_{i-1}}$. If the feedback is negative then strict convergence requires that α_{i-1} be changed such that $F_{\alpha_i}(\mathbf{x}_i)=\bar{F}_{\alpha_{i-1}}(\mathbf{x}_i)$.

In a different approach the teacher presents the desired goal value $y_i=G(\mathbf{x}_i)$ as a *feedforward* signal along with each example [Alek79, Arms78]. The central controller takes immediate action by changing the assignment, if necessary. Both approaches (feedback or feedforward) are in theory equivalent.

The examples are sometimes taken directly from a real-world environment, and then neither the learner nor the teacher have control over the sequence order. Sometimes the teacher selects the examples, and then there is a training phase entirely separate from the use in a real-world environment. The latter approach is more efficient in that the length of the sequence necessary to transfer a certain goal function can be minimized by properly selecting the set and the order of the examples. Finally, the learner itself is sometimes the source of the examples, which are then called *oracles* [Vali84]. In what follows we will assume that the teacher provides the examples.

Three strategies for achieving convergence towards the goal function are described below [Alek68b].

a. Search in the assignment space

In this strategy the central controller tries all possible combinations of node assignments one by one in some arbitrary order. The assignment remains unchanged until an example incompatible with the current network function is presented. When this occurs the central controller tries the next assignment. The error signal therefore triggers a sequential search through the set of assignments. The search space is of size $P=p^l$. If there exists an assignment that implements a network function compatible with the goal function G_L, then eventually it will be found since the controller tries *all* assignments one by one.

This strategy does not have the property of strict convergence. Indeed, changing the assignment may not classify previous examples (or even the current example) correctly. It is a blind exhaustive search. As a result, the number of examples needed to reach a correct assignment is typically large.

b. Search in the network function space

An improvement in efficiency can be achieved if the controller searches directly in the space of the network functions. When a discrepancy with an example occurs the controller takes a new network function and generates the appropriate assignment. The controller must therefore be able to determine all network functions and a correct assignment for each of them. The search space is now of size $Q=|\Phi|$. This approach also lacks strict convergence as it does not take into account previous examples.

c. Search in the input space

By far the most efficient strategy for learning a Boolean function is to integrate the goal values of all examples shown into a partial goal function $G_i(\mathbf{x})$. In this way a complete goal function can be specified with only $\log_2 Q$ examples. For any goal function there are only 2^n different examples, namely the 2^n entries in the truth table of the goal function. Showing all examples just once conveys the goal function to the perceptron.

This strategy has the property of strict convergence, but it requires more work from the central controller than the other strategies. The network function must always correctly classify the current example and also all previous examples. We will discuss some examples of this strategy later in this chapter.

5.1.4. Perceptrons with Distributed Control

In a perceptron with distributed control each node has a small local controller that selects its function. A 2-input node with its local controller is represented schematically in figure 5.1. The wide arrow represents a global communication bus, used for instance to broadcast the feedback signal from the environment, or used by the controllers to communicate with each other. Each controller has access to the local input and output values of the node, and has a certain amount of local memory.

A perceptron with distributed control is an interconnection of such nodes and therefore a network of small finite state automata. This network is

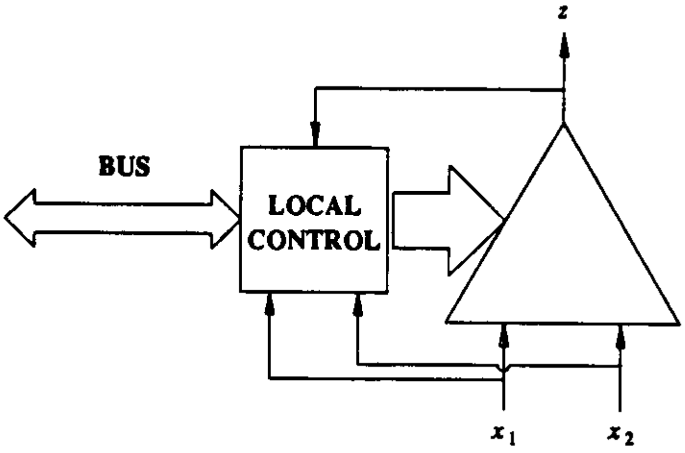


Figure 5.1: Schematic representation of a 2-input node of a perceptron with its local controller.

used as follows. Each example is presented to the global input lines of the perceptron and causes a computational flow through the data path. Each controller receives a processed piece of the example, namely the local inputs to its node. Other information that the controller might need must be passed between the controllers directly.

Three desirable characteristics of distributed control are reviewed below.

a. Limited local memory

It is desirable to keep the size of the local memory used by each controller as small as possible. A typical requirement is that its size be independent of the size of the network. In other words, adding more nodes or inputs to the network should not increase the size of the required local memory. Enhancing the capability of the network is achieved by increasing the number of nodes, not the local memory at each node.

b. Distributed memory

It is also desirable to distribute only limited information available from a single example. Figure 5.1 shows that each controller knows the local inputs and output for the node, but it does not know the input values to other nodes in the network. This implies that a controller can decide in which column of the decomposition chart of the goal function the current minterm example is located, but it does not know its location inside the column.

c. Limited communication

Limiting the communication between the controllers or between the controllers and the environment would also be important in a practical system. Typically, only a feedback or feedforward signal is allowed on the global bus, and the controllers do not use this bus to communicate with each other.

The following three requirements represent *extreme* distributed control:

1. The size of the local memory is independent of the size of the network.
2. The controller has no knowledge of the input values of other nodes.
3. The global communication is restricted to a feedback or feedforward signal.

Such extreme distributed control is present in all the extant research on learning by example in perceptrons, but it is a strong constraint on the system. We will assume the same constraints in the remainder of this chapter.

5.2. THRESHOLD GATE PERCEPTRONS

The number of adjustable nodes in a perceptron determines not only the functional completeness of the system and the complexity of the decomposition problem, but also, and even more so, the complexity of the problem of learning by example. Below we review the work on learning by example in threshold gate networks using the number of adjustable nodes as the basis for the classification. The *simple perceptron* (a single-node system) is discussed first, followed by single-layer perceptrons, and finally multilayer perceptrons.

5.2.1. The Simple Perceptron

The learning scheme used in the *simple perceptron* is straightforward because only one threshold gate is involved in the learning. Below we review the basic algorithm; for a more complete treatment we refer to [Duda73, Bow84].

Consider the R-unit of a *simple perceptron* (figure 2.11) and label its inputs $\mathbf{x} = (x_1, \ldots, x_k)$. Its output is a linear threshold function of its inputs:

$$z = [\sum_{j=1}^{k} w_j x_j \geq \theta] = [\mathbf{w}^t \cdot \mathbf{x} \geq \theta]$$

Each example shown to the perceptron produces an input vector $\mathbf{x}_i$ to the R-unit. Call $\mathbf{w}_i$ and θ_i the values of the weights and threshold of the R-unit when the i-th example is presented. The perceptron then produces an output

$$z_i = f_i(\mathbf{x}_i) = [\mathbf{w}_i^t \cdot \mathbf{x}_i \geq \theta_i]$$

Assume that in the beginning of the learning phase the values $\mathbf{w}_0$ and θ_0 are chosen arbitrarily.

For each example shown, the teacher compares z_i with the desired value and issues a negative feedback signal if necessary. When the control of the R-unit receives a negative feedback it changes the threshold function in the following way:

$$\begin{cases} \mathbf{w}_{i+1} = \mathbf{w}_i + \delta_i \mathbf{x}_i \\ \theta_{i+1} = \theta_i - \delta_i \end{cases}$$

where $\delta_i = 1$ if $\mathbf{x}_i \in C_1$ (the desired response is 1) and $\delta_i = -1$ if $\mathbf{x}_i \in C_0$ (the desired response is 0). If no negative feedback occurs then the threshold function is not changed. This scheme for updating the values of $\mathbf{w}$ and θ is called the *fixed increment rule.*

THEOREM 5.1: [Rose62b, Bloc61, Bloc62a] If a set of inputs $\{\mathbf{x}_i\}$ presented to the R-unit is compatible with a linear threshold function and if each of them is shown equally often, then the fixed increment rule will converge to values for **w** and θ that correctly classify all the examples shown. □

Stated differently, this theorem says that, if the examples are shown often enough then the learning strategy will find a threshold function compatible with the given set of inputs, if one exists. Historically this theorem has been called the *perceptron convergence theorem* [Sing62], but the name *threshold gate convergence theorem* would have been more appropriate. The theorem says nothing about the examples that were not presented. Furthermore, nothing is known a priori about the length of the example sequence since it depends on the sequence order.

The fixed increment rule does not have the property of strict convergence. A generalization of this rule is called the *variable increment rule:*

$$\begin{cases} \mathbf{w}_{i+1} = \mathbf{w}_i + \rho_i \delta_i \mathbf{x}_i \\ \theta_{i+1} = \theta_i - \rho_i \delta_i \end{cases}$$

where ρ_i is a positive scalar that differs from step to step. For a particular choice of ρ_i strict convergence can be assured, that is, the current and all previous examples will be classified correctly by the new threshold function (at least if the examples are compatible with a threshold function). We refer to [Duda73, Bow84, Gall85a] for details on how to chose ρ_i.

The convergence theorem is independent of the nature of the inputs (analog or digital); digital inputs are considered a special case of analog inputs. In any perceptron, including the *simple perceptron,* the inputs to the nodes are always binary values because they are connected to the outputs of other nodes. However, the theory also applies to the *Adaline* [Widr60], which is a single threshold gate with analog inputs.

Variations on this error-correction scheme appeared in the early perceptron literature. For example, variations of the fixed increment rule were proposed by Rosenblatt and received names such as *uncompensated gain system, constant feed system, parasitic gain system* [Rose58]. More fundamentally different was a proposal to change the R-unit even when there is no negative feedback, called *forced learning* [Bloc62a]. A variation of the variable increment rule was used, namely the sign of ρ_i is inverted if the feedback is positive. Convergence can still be proven in this more general case.

5.2.2. Single-Layer Perceptrons

Most of the later work in learning perceptrons concerns networks with a single layer of adjustable nodes. We review three of the most important prototypes, although many others exist.

The first structure is the *Madaline* proposed by Widrow [Widr62, Widr64]. A Madaline is a network of *Adalines* (threshold gates). A layer of threshold gates is connected to the inputs and the outputs are connected to a voting gate (figure 5.2). The voting gate produces an output 1 if more than half of its inputs are 1 (all the weights are fixed to 1 and the threshold is equal to half the number of inputs). The learning proceeds as follows. If the output is correct for a given example then no change takes place. If the output is incorrect then the fixed increment rule is applied to the gates with the highest sum before thresholding ($\mathbf{w}_i^t \cdot \mathbf{x}_i$) and to as many threshold gates as needed to correct the output. Widrow states that one of his doctoral students proved the convergence of this scheme, but no reference is available.

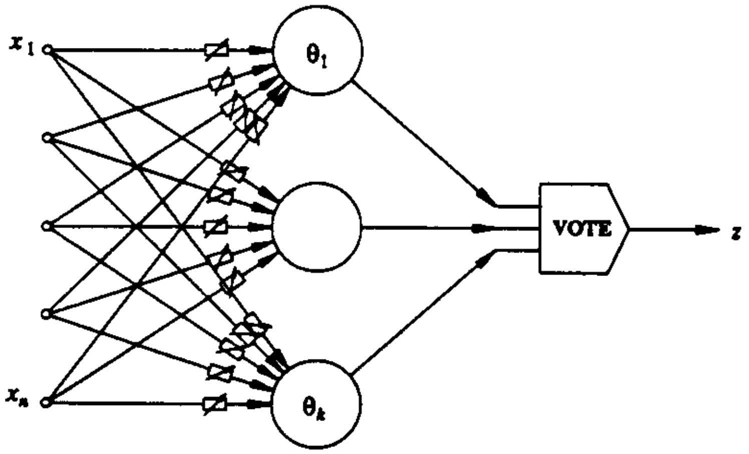


Figure 5.2: Structure of a Madaline.

A similar approach was developed by Takiyama in a perceptron called a *committee machine*. (figure 5.3). This perceptron is like a Madaline except that the weights and threshold of the output node can be different from those of a voting gate [Taki78]. Alternatives include an AND gate or an OR gate. Takiyama developed and proved a general learning strategy that can be adapted easily to the particular values of the weights and threshold of the top gate. The strategy is similar to Widrow's scheme except that the fixed increment rule is applied to *all* threshold gates of the first layer.

A *two-level committee machine* consists of one layer of adjustable

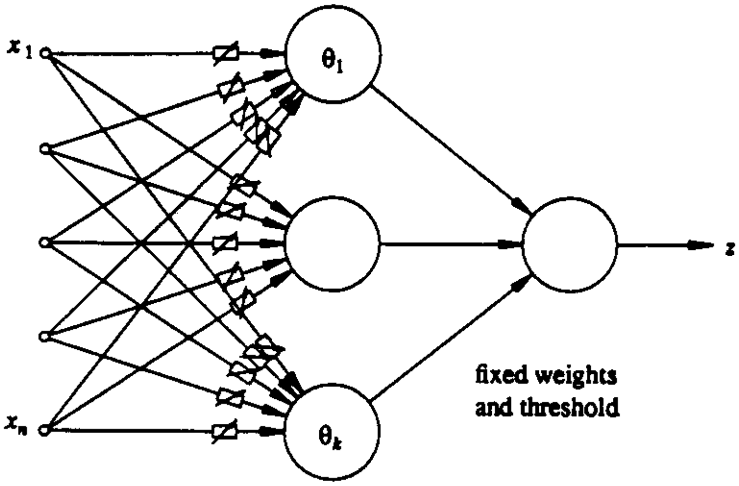


Figure 5.3: Structure of a committee machine.

threshold gates followed by two layers of fixed threshold gates (figure 5.4) [Taki81]. The latter can take different forms, such as voting gates, AND gates, OR gates, etc. Again, the learning algorithm is derived from the fixed increment rule.

5.2.3. Multilayer Perceptrons

A formal analysis of learning by example in perceptrons with more than one layer of adjustable nodes is difficult. Widrow proposed to adjust the nodes in the bottom layer more frequently than those in higher layers [Widr62, Widr64]. A similar approach was used by Rosenblatt, who also assumed a certain order in the sequence of examples [Rose62a]. These heuristic learning strategies converged for the experiments discussed in these

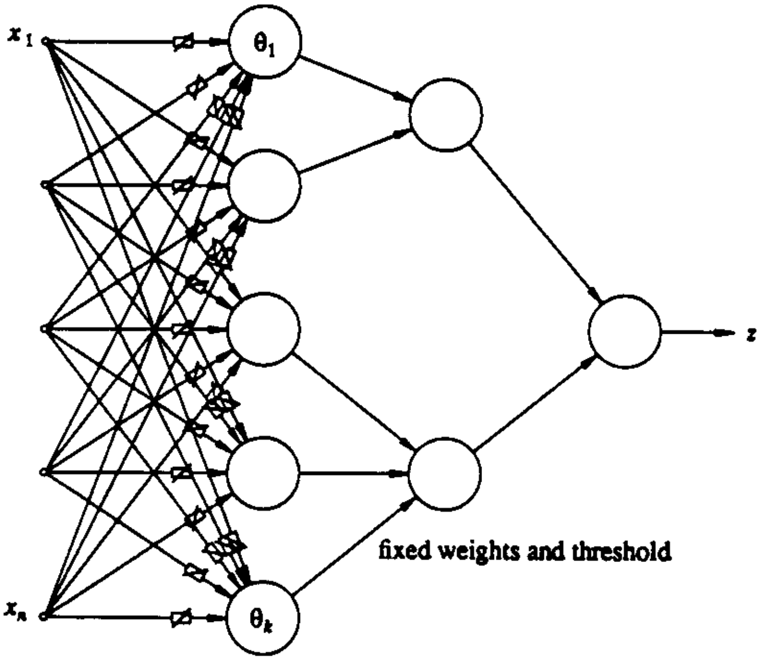


Figure 5.4: A two-level committee machine.

publications, but a theoretical treatment was not given and indeed is missing in all other research on learning in multilayer perceptrons. The strategy of *annealing* used in the *Boltzmann machine* is one example of a formally analyzed scheme for learning by example in a general class of threshold gate networks [Hint84]. This strategy, based on thermodynamic principles, converges even for multilayer networks. However, it is extremely inefficient because it requires many iterations and much internode communication for each example shown.

5.3. DIGITAL PERCEPTRONS

Implementations of digital nodes were described in section 2.1.3.1. In these models, it is straightforward to change the local function such that a given input produces a particular output. For a RAM or a ULM node (figure 2.6) this is achieved by setting the appropriate truth table entry to the desired value. For an incomplete node (for instance figure 2.7) the change may not always be possible. For the node of figure 2.7, for instance, the two control bits c_1 and c_2 represent two different entries in the truth table, as illustrated by its Karnaugh map in figure 5.5. A given input can therefore not always produce any output value.

		x_1	
		0	1
x_2	0	0	c_2
	1	c_1	1

Figure 5.5: Karnaugh map of the incomplete node of figure 2.7.

In summary, the function-set model makes local assignment straightforward and we can directly address the responsibility assignment problem in networks.

This section discusses three distributed implementations of learning by example using a search in the input space. Two approaches are taken from the literature and are discussed first, followed by our contribution. The resulting observations and conclusions apply to all three as well as to many other perceptron learning strategies. The first approach allows any order in the sequence of examples, while the second and third impose certain constraints on this order. On the other hand, the first approach applies only to single-layer perceptrons, while the second and third use a disjunctive binary tree with respectively incomplete nodes and arbitrary nodes. In other words, in this section the generality and complexity of the perceptron structure is inversely related to the generality of the input sequence order.

In what follows we assume that the perceptron can implement the given goal function G_L. Furthermore, we assume that *all* 2^n examples are shown (G_L is completely specified) and, as a result, no generalization is necessary. Ultimately, generalization must be an integral part of learning by example, and future research should extend the work reported here to include generalization.

5.3.1. Single-Layer RAM Networks

This section reviews work that started around 1965 by Aleksander at Brunel University in the UK. Aleksander was the first to suggest a digital model as a replacement for the classical threshold gate model [Alek68b]. Later, commenting on Yau and Tang's paper on ULMs [Yau70], Aleksander saw ULMs as a possible implementation of this digital model and as a practical building block for *adaptive logic systems* (perceptrons) [Alek71]. More recently, he has turned to Random Access Memories (RAMs) as the building block for learning systems [Alek79].

The operation of a RAM is illustrated in figure 5.6. It takes n inputs and holds 2^n bits. Each input pattern addresses one bit of the memory. In the data processing mode, the value stored at this address is directed to the output z. The operation is then a table lookup of a truth value. In the learning mode, the value of the TEACH input is written at the address specified by the current input pattern. The entries in the truth table that correspond to examples not yet shown remain arbitrary. Viewed as a functional node, a RAM is complete and can implement any Boolean function of its inputs.

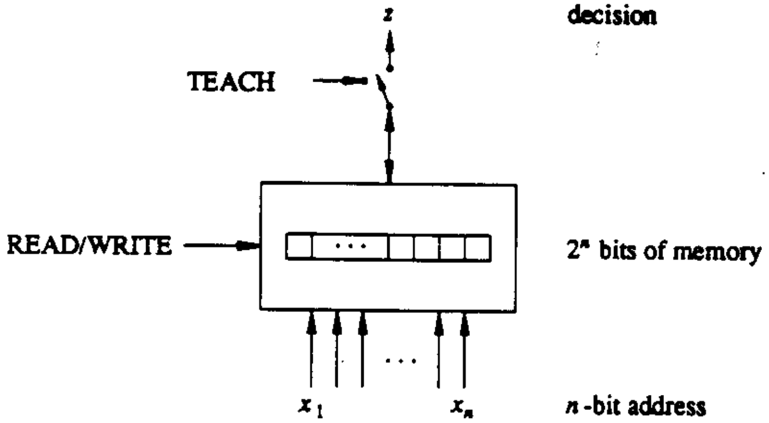


Figure 5.6: The operation of a RAM as the basic node of a perceptron.

Aleksander's single-output network is shown in figure 5.7. It is a network in which a layer of RAMs is connected to a fixed gate (typically an AND gate, an OR gate, or a voting gate). All RAMs receive the same signal from the teacher, which constitutes the only global communication. The

RAMs are totally independent of each other and their local control is the usual READ/WRITE circuitry present in any conventional RAM.

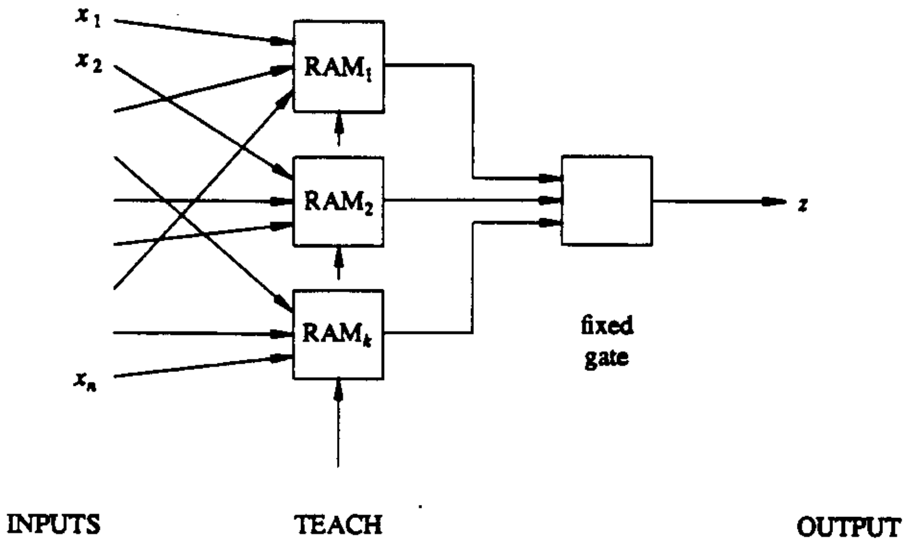


Figure 5.7: Aleksander's single-output network.

The external inputs are connected to the address lines of the RAMs, and these connections can be done either randomly or structured to suit the intended application. Structured input connections are shown in figure 5.8, where the inputs are taken from a 3×3 photo cell array.

In this figure the network is a disjunctive tree and it is incomplete. Figure 5.9 illustrates this incompleteness. The two input examples shown, which have the same output, are used to train the system. As a result, the third input pattern always produces the same output as the two examples, regardless of what output is desired. In other words, if the first two examples

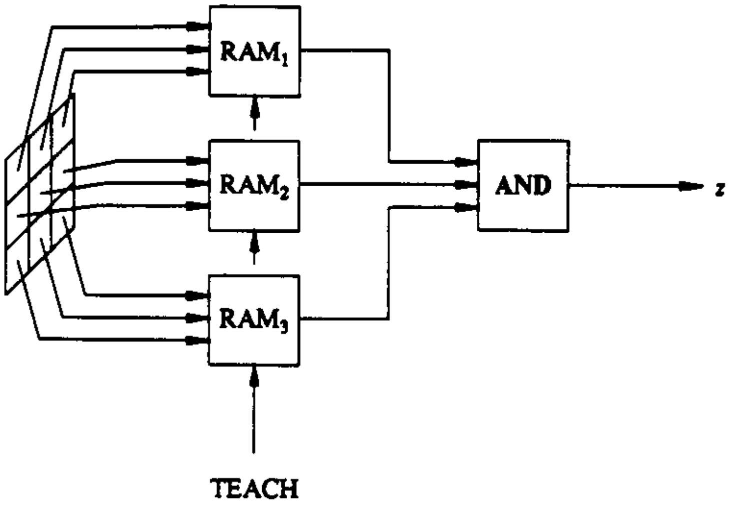


Figure 5.8: An example of structured input connections in a RAM network.

have an equal output value then the third example must have the same output or the task cannot be implemented. A brief analysis of this incompleteness issue is given in [Ullm69, Alek70a, Alek84a].

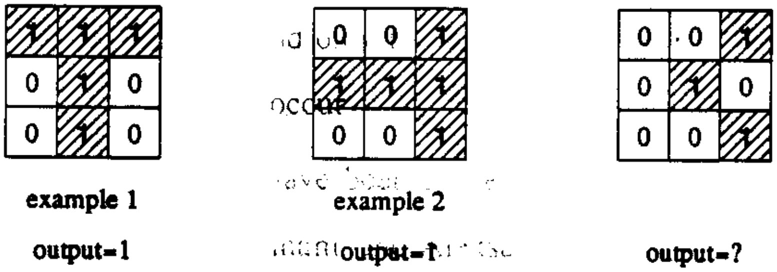


Figure 5.9: Example of incompleteness in the RAM network of figure 5.8.

We now review how the transfer of the goal function occurs in this network. First assume that the top node implements an AND function [Alek79] and consider an example with desired output 1. Since the top node is an AND gate, the outputs of *all* the RAMs must be 1 if the system is to behave correctly. Hence, for each example of class C_1, the corresponding entries in the RAMs must be 1. The responsibility assignment is straightforward, namely *all* nodes in the bottom layer are responsible.

On the other hand, if an example of C_0 is shown then at least one RAM should output a 0. If none output a 1 then *at least one* of the addressed bits must be changed from 1 to 0, but which one? No obvious decision can be made since it depends on the input connections and the particular goal function. Assignment of responsibility is not possible without more information.

One approach assumes that the RAMs initially contain nothing but zero entries, that is, the nodes initially implement the always-zero function. It is then never necessary to switch a bit from 1 to 0 since all zeros are already present and those that change to a 1 never have to be changed back.

A different approach assigns responsibility for a zero output to *all* the nodes. Consequently, in the early phase of the training too many zero entries are created, but these are changed to 1 by the examples in C_1. In other words, the initial learning phase directs the RAMs towards a zero initial state, after which the proper learning can proceed.

To summarize, in each learning step the value of the desired output (either 1 or 0) is written into *all* the RAMs at the addresses specified by their local input. If initially all the RAMs contain only zeros, then this strategy assures strict convergence and only examples of C_1 (at most 2^n) are needed to attain the goal function. If initially the RAMs are arbitrary then more examples will be necessary (but less than 2×2^n). If the examples of C_0 are shown first followed by the rest then 2^n different examples are always enough, regardless of the initial state. These properties hold independently of the particular arrangement of the input connections.

If the output node implements an OR function then all the nodes are responsible for each example of C_0, whereas responsibility for C_1 depends on the input connections and the particular goal function. The learning algorithm is identical to the case with an AND gate except that the preferred initial state is the complement. If the RAMs are not in this initial state then showing the examples of C_1 first improves learning speed.

If the top node implements any other (nontrivial) function (for instance a voting gate) then direct assignment of responsibility is not possible. One could for instance assign responsibility to *all* the nodes or to a randomly chosen subset, but this process will typically cycle and only for special orderings of the sequence of examples will convergence occur.

This simple structure and its learning strategy have been implemented as a hardware device called *WISARD* (''Wilkie, Stonham, and Aleksander's Recognition Device'') [Alek83a]. The device has an array of 512×512 inputs and a layer of 32,768 RAMs of 256 bits each (8-bit address), totaling 8 Mb of storage. The input connections are disjunctive. The design has been used successfully for recognizing facial pictures projected onto a 512×512 array of photocells [Alek83b], although no specifics of these experiments are given.

5.3.2. Disjunctive Binary Trees with Incomplete Nodes

This section reviews the work by Armstrong at the University of Montreal in Canada [Arms79]. The node used in this work is a 2-input Boolean gate that implements only 4 functions (γ=0.25), as discussed in section 2.1.3.1. The physical implementation of the node is shown in figure 2.7. The nodes are interconnected to form a disjunctive binary tree network (see for instance figure 5.10).

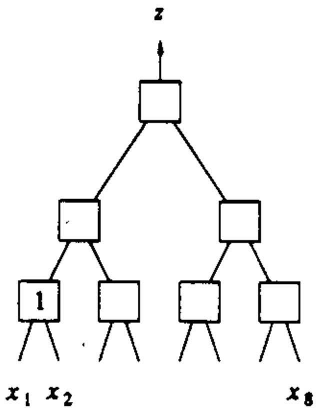


Figure 5.10: Example of a disjunctive binary tree network.

Each node implements the *nonconstant increasing functions*. An *increasing* function is a function that never changes its output from 1 to 0 when one of its inputs is changed from 0 to 1 (the output either changes from 0 to 1 or remains unchanged). A *nonconstant* function is one that is not always zero or one. The class of nonconstant increasing functions is very incomplete. For 2 inputs, it contains only 4 functions (table 2.2): {AND,OR,LEFT,RIGHT}.

A disjunctive binary tree has the property that, if all its nodes implement a nonconstant increasing function, then the network also implements a nonconstant increasing function [Arms79]. Hence, the networks considered by Armstrong always implement a nonconstant increasing function.

Another property of increasing functions that will be used shortly is the following. If $F(x_1, \ldots, x_n)$ is an increasing function then the four residual functions

$$F(x_1, \ldots, x_i{=}0, \ldots, x_j{=}0, \ldots, x_n)$$

$$F(x_1, \ldots, x_i{=}0, \ldots, x_j{=}1, \ldots, x_n)$$

$$F(x_1, \ldots, x_i{=}1, \ldots, x_j{=}0, \ldots, x_n)$$

$$F(x_1, \ldots, x_i{=}1, \ldots, x_j{=}1, \ldots, x_n)$$

$(\forall i,j)$ are also increasing functions. The columns of the decomposition chart for node 1 of figure 5.10, shown in figure 5.11, represent the residual functions after fixing x_1 and x_2 to the values 00, 01, 11, and 10 respectively. Since the network always implements an increasing function, these columns represent four increasing functions.

Transfer of knowledge in this network proceeds as follows. All the

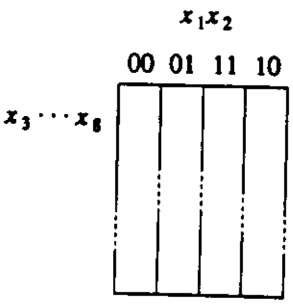


Figure 5.11: Decomposition chart for node 1 of the network of figure 5.10.

examples are shown to the network, in no specific order, and the associated goal values are broadcast to the nodes in the bottom layer. The local functions of these nodes are not changed immediately, but local controllers accumulate the information. When all examples have been shown, the control units in the bottom layer assign local functions to their nodes. The cycle then repeats: all examples are shown again and the teaching signals are sent to the nodes in the next layer, and so on until the top node has received its assignment. The transfer of knowledge therefore proceeds in a bottom-up fashion. A binary tree with n inputs has $\log_2 n$ layers and therefore each example must be shown $\log_2 n$ times. This implies an example sequence of length $2^n \log_2 n$. The order of the example sequence within each cycle is arbitrary.

The assignment decision at each node is based on the decomposition chart and the knowledge that the network implements an increasing function. Consider node 1 of the network in figure 5.10 and the decomposition chart of a goal function (figure 5.11). There are at most 2 distinct columns in this decomposition chart and the local function must be derived from these column differences. When an example is shown, the column of the decomposition chart it belongs to is specified by the local input to the node, but the inputs to other nodes are not known locally. Since the goal function G is an increasing function, it follows that

$$\forall \mathbf{s} \in \{0,1\}^{n-2} : G_{00s}=1 \Rightarrow G_{01s}=1 \text{ and } G_{10s}=1 \text{ and } G_{11s}=1$$

$$\forall \mathbf{s} \in \{0,1\}^{n-2} : G_{01s}=1 \Rightarrow G_{11s}=1$$

$$\forall \mathbf{s} \in \{0,1\}^{n-2} : G_{10s}=1 \Rightarrow G_{11s}=1$$

This means that a difference in columns implies a difference in the number of 1's and 0's in these columns, and vice versa. Hence, the discrepancy between columns can be deduced from a count of the number of 1's or 0's. Denote the number of 1's in each column O_{00}, O_{01}, O_{10}, and O_{11} respectively. Since the goal function is increasing, it follows that

$$O_{00} \le O_{01} \le O_{11}$$

$$O_{00} \le O_{10} \le O_{11}$$

The control bits c_1 and c_2 (the assignment) are selected based on whether O_{01} and O_{10} are equal to O_{00} or O_{11}. If O_{01} is equal to O_{11} then both columns should take the same functional value; this implies $c_1=1$ since the value of the 11-column is always 1. Otherwise $c_1=0$. The same holds for c_2 with respect to O_{10}. Therefore:

$$c_1 = \begin{cases} 1 & \text{if } O_{01}=O_{11} \\ 0 & \text{otherwise} \end{cases}$$

$$c_2 = \begin{cases} 1 & \text{if } O_{10}=O_{11} \\ 0 & \text{otherwise} \end{cases}$$

For nodes in higher layers of the tree the same strategy can still be used, but now the *fraction* of observed ones and zeros must be used:

$$c_1 = \begin{cases} 1 & \text{if } \dfrac{O_{01}}{Z_{01}}=\dfrac{O_{11}}{Z_{11}} \\ 0 & \text{otherwise} \end{cases}$$

$$c_2 = \begin{cases} 1 & \text{if } \dfrac{O_{10}}{Z_{10}}=\dfrac{O_{11}}{Z_{11}} \\ 0 & \text{otherwise} \end{cases}$$

Z_{00}, Z_{01}, Z_{10}, and Z_{11} are the number of zeros in each column. We refer [Arms79] for a proof.

Each local controller needs a small amount of memory to count the occurrences of zeros and ones in each column of the decomposition chart (4×2×n bits per node). All the controllers work independently and the global communication is restricted to the presentation of examples and goal values.

The approach described here has been implemented in hardware [Arms71, Arms76]. In one application for LANDSAT image processing a 32-input tree was built [Arms78].

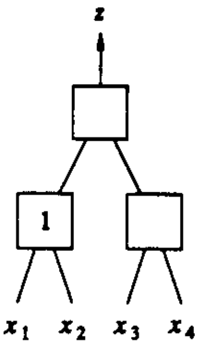
5.3.3. Disjunctive Binary Trees with Arbitrary Nodes

Instead of restricting the node functionality to nonconstant increasing functions, consider the general case with arbitrary nodes (complete or incomplete). The approach we develop below uses only binary trees, but is valid for any disjunctive tree network.

The strategy is again based on the local decomposition chart for each node. Since the nodal functions are arbitrary, a count of the number of ones and zeros does not determine which columns of the decomposition chart are different. Yet, this knowledge is the basis for responsibility assignment. How can a node decide column differences?

Consider the decomposition chart for node 1 of a 3-node 4-input disjunctive binary tree (figure 5.12). In this example, the number of ones and zeros is identical for all columns, namely 2. Hence, this count does not provide any information regarding the column differences. Furthermore, considering *two* separate examples does not provide any information either. For instance $\mathbf{x}_i$=(0,0,0,0) and $\mathbf{x}_{i+1}$=(0,1,0,1) have different goal values, yet the columns are equal.

By contrast, a difference in goal value for two examples with the *same* values for x_3 and x_4 implies a difference in the corresponding columns, and therefore a difference in local function value. Consider the input pair $\mathbf{x}_j$=(0,0,0,0) and $\mathbf{x}_{j+1}$=(1,0,0,0). Since in figure 5.12 $G(\mathbf{x}_j) \neq G(\mathbf{x}_{j+1})$, it follows that the corresponding columns are different, and this implies $f_1(0,0) \neq f_1(1,0)$. On the other hand, the equality of the goal value for a pair



x_3x_4 \ x_1x_2	00	01	11	10
00	0	0	0	1
01	1	1	1	0
11	0	0	0	1
10	1	1	1	0

Figure 5.12: Example of a 4-input disjunctive binary tree network and the decomposition chart for node 1 of a simple goal function.

of input examples does not imply the equality of the corresponding columns of the decomposition chart. For example, in figure 5.13 the two inputs $\mathbf{x}_i$=(1,1,0,0) and $\mathbf{x}_{i+1}$=(1,0,0,0) have the same goal value, yet their columns are different.

x_3x_4 \ x_1x_2	00	01	11	10
00	0	0	0	0
01	1	1	1	0
11	0	0	0	0
10	1	1	1	0

Figure 5.13: Two examples from different columns may have the same goal value.

In summary, the local controller can make a valid conclusion based on the current and previous examples provided the current example differs from the previous one in its local input variables only, and provided they belong to a different class.

One approach therefore imposes a strict order on the sequence of examples. A pair of consecutive examples should have different bits only for the input variables connected to the same node. For instance, the sequence could be such that two consecutive examples are a Hamming distance of 1 from each other. Each node in the bottom layer can then locally and independently decide whether it is responsible for a difference in goal value. If enough pairs of examples have been shown then each node can determine a local function (up to a negation).

As in the previous section, the transfer of knowledge proceeds by training the layers one by one. First the nodes in the bottom layer are taught the correct assignment; the sequence of examples is then shown again to teach the next layer; and so on to the top node. As the training proceeds upwards in the tree, fewer examples suffice. For the top node only 4 examples are needed. However, *which* examples are needed depends on the assignments made to nodes lower in the tree. As the teacher has no knowledge of the internal structure, these assignments are not known and hence neither is the optimal order of examples. We will therefore assume that in each cycle the same sequence of examples is shown.

The learning proceeds layer by layer; assignments to a node can be done only if all its child nodes have received the correct assignment. Indeed, the pairs of examples do not give information regarding nodes in different layers, or even different nodes in the same layer. Hence, the incremental transfer of knowledge only accrues to one node at a time.

A different strategy uses an explicit teach signal to start and stop the knowledge transfer. Each time an example differs from the previous one in more than a single bit, the transfer process is stopped. This is a more flexible approach, but the perceptron still learns only when the examples differ in not more than one bit.

The total number of pairs of n-input vectors that differ in only one bit is $n \times 2^{n-1}$. The number of examples needed to train one layer is therefore at most $n \times 2^n$. This implies $n 2^n \log_2 n$ example presentations for the whole network, which is larger than the $2^n \log_2 n$ examples needed in the strategy of the previous section. In spite of the ordering of examples, the learning is slower.

The local controllers work independently of each other, except for a global teach signal and a signal that enables the layer being trained. Each node stores the previous and the current local input, as well as which columns are different. The latter requires 2 bits per column (total 8 bits); the former requires 2×3 bits (2 bits for the example and 1 for its goal value). Hence, 14 bits of memory are required per node, independently of the size of the network. Compare this with the $8n$ bits needed in section 5.3.2. This suggests that Armstrong's strategy is more efficient in the number of examples it needs at the expense of more memory requirements.

5.4. CONCLUSIONS

Aleksander discovered that in single-layer networks with an AND gate or an OR gate as the output gate a simple assignment of responsibility assures convergence. Armstrong discovered a class of networks that allows a correct assignment of the responsibility based on a local counter. However, small extensions to both classes, such as for instance a voting gate as the output node in Aleksander's network or slightly more complete nodes in Armstrong's tree network, cause the strategies to collapse. A direct extension of these schemes to more general networks is not possible. Success here is due to specific restrictions on the class of perceptrons.

What can be done in the general case? A first observation is that pairs of examples give more information than single examples, especially if they differ only in one bit. This allows a node with a difference in input variables to assume that all the other variables of the pair are identical. This approach works only if the teacher has control over the order of the examples. Nevertheless, it makes intuitive sense if we compare it with the way humans often teach new concepts, namely by showing examples and counterexamples that differ only in one or a few distinguishing features.

The structure of the interconnections is again extremely important. We studied only disjunctive binary trees; with fan-out connections the local assignments are interdependent and not unique. The complexity of the learning strategy is related to the completeness and (nontrivial) redundancy of the network. The precise nature of these relationships are currently still unknown.

Incremental assignment of responsibility in a perceptron with distributed control is harder than in the decomposition problem. One reason is the restricted and distributed nature of the allowable memory. Yet, these requirements are fundamental in perceptrons, either with a threshold gate model or a digital model, because of their practical advantages.

Most of the existing work in perceptrons (for instance [Hebb49, Klop72, Klop82]) assumes that extreme distributed control can result in a globally desirable function. In other words, it is conjectured that globally interesting properties can *emerge* from a collection of small nodes whose operation is guided by strategies that are entirely local. It follows from our work that with extreme distributed control convergence cannot be guaranteed in general. On the contrary, each node must have some global information to guide its local operation. For instance, Fukushima proposes a learning strategy in which each node communicates with a group of nodes in the same layer [Fuku75, Fuku80]. Simulation experiments show a positive result, although no formal proof is given.

Much work on learning by example in perceptrons makes random changes in the assignments with each negative feedback; it is hoped that randomness will imply proper, if perhaps slow, convergence. Typical examples are [Alde75, Mart83]. Imposing constraints on the order of the examples is the alternative strategy. The additional information in the order of the examples allows more informed local decisions at the nodes and hence a more efficient strategy. These order constraints, however, are imposed by the particular internal structure of the perceptron.

CHAPTER 6

CONCLUSIONS

This dissertation has developed some initial steps towards an integrated study of perceptrons, particularly the problem of assigning functional responsibility. An abstract Boolean treatment was used and hence the results are independent of a physical implementation.

A summary of the contributions is given first, followed by some directions for future research.

6.1. CONTRIBUTIONS

6.1.1. Perceptrons

Perceptrons have been traditionally identified with specific and restricted networks of threshold gates. Chapter 2 argued the existence of a larger class of perceptrons and showed how the perceptron integrates a broad field of research.

We defined perceptrons as multilayered networks of polyfunctional combinational nodes. Central to this definition is the fact that the functional flexibility is not restricted to a single node or a single layer of the network.

The fundamental problem addressed in the research is assignment of functional responsibility in multilayered networks. In particular, our work identifies two important properties that form the central problem for the responsibility assignment in perceptrons: multilayeredness and fan-out connections.

We presented an overview of existing and proposed implementations of perceptrons, both analog and digital, and developed a model that integrates them. In our model, different levels of completeness and redundancy can be achieved, depending on the specifics of the node and the interconnections.

6.1.2. Applications

Chapter 3 showed that the domain of possible applications for perceptrons covers an area larger than typically assumed in the literature. The combinational nature of the perceptron is suited to problems that can be expressed as Boolean functions.

We showed how a subset of rule-based inference systems can be reduced to Boolean functions and implemented with perceptrons. We also showed that expressing a rule base as a pair of complementary Boolean functions can usefully expose incompleteness and inconsistencies in the rule set.

The characterization and preliminary study of this application domain is the second contribution of our work.

6.1.3. Assignment of Functional Responsibility

The central issue in perceptron applications is the functional responsibility assignment in the multilayered network. Hence, the most significant contributions of our work are related to this issue.

The problem of responsibility assignment can manifest itself in many different forms. We first studied in some detail an extreme form of learning, called decomposition. Steps towards learning by example were discussed subsequently. Introducing decomposition as a reference to discuss learning in perceptrons has provided new insights into the problem.

Chapters 4 and 5 applied the theory of decomposition of Boolean functions to the learning problem and exposed what makes responsibility assignment such a hard task. Perceptron networks have both depth and width. In isolation neither of them is a source of difficulty, but jointly they determine the degree of complexity.

The three most important conclusions are reviewed below.

a. Fan-out connections

If a perceptron has fan-out connections, that is, if an input to the network or the output of an internal node is connected to more than one node, the network becomes redundant in a nontrivial way. As a result, the assignment of responsibility is no longer unique. Furthermore, the local functional assignments to different nodes of the network are not independent of each other. Therefore, an assignment of responsibility based solely on local information is not possible.

With no fan-out connections present (a disjunctive tree) responsibility assignment becomes straightforward and can be done in an efficient way.

b. Decomposition

We approached the decomposition problem with a search algorithm. The nodes are treated one by one in a bottom-up fashion. At each node a few promising assignments are selected by a local selection criterion and tried in sequence. A reduction process between two consecutive nodes causes the assignments selected for a given node to depend on the assignments made to previous nodes. The difficulty of the search is that this local selection of assignments should also depend on the remainder of the perceptron structure, the part that has not yet been considered by the search. This is the underlying pitfall of the responsibility assignment problem, namely, a correct decision at each node requires some global knowledge of the network.

c. Learning by example

The same discrepancy between local and global knowledge occurs in learning by example, which has also been briefly addressed in this dissertation. We presented a simplified approach for binary tree networks that treats the nodes layer by layer and makes assignments to one node at a time. Therefore, with a multilayered network the learning remains confined to a single node at a time.

6.2. FUTURE RESEARCH

We view our work as an initial step towards a formal study of perceptrons. Only a few issues in a large and difficult area were explored. The questions that remain unanswered are numerous and many directions for future research can be proposed.

6.2.1. Perceptrons

We have unified a broad class of structures and collectively labeled them *perceptrons*. Detailed studies of the various alternatives and possible trade-offs in this wide spectrum of structures are needed. Specifically, little is known about the functional capabilities and redundancy of multilayered systems. For instance, the following questions cannot be answered at this time. How does one determine the functional set Φ of a given perceptron? How does one design a network that implements a given set Φ? Is there a simple way to determine whether a given network can implement all the functions in a set Φ? How many different ways can a given function be implemented on a given perceptron? How can redundancy be exploited for fault tolerance?

The underlying problem is our present ignorance regarding the relationship between the structure of a multilayered network and its functional capabilities. Existing tools, such as conjunctive or disjunctive normal form, Karnaugh maps, and the like, are not appropriate for perceptrons.

6.2.2. Applications

Further research on the topic of combinational rule-based systems should explore the rule-based inference systems that can be expressed with Boolean logic. The restriction of propositional logic is too strong. Certain rule bases expressed in predicate logic can also be converted to a Boolean formalism. The specific methods and constraints for this transformation have to be developed and analyzed. Considering the improvement in efficiency, this area of research should be of significant importance for the near-term future.

6.2.3. Decomposition

Three important aspects of the decomposition problem are open for future research.

a. Existing decomposition algorithm

Many aspects of the decomposition algorithm, as described in chapter 4, need to be explored to improve not only the performance of the algorithm itself, but also provide more insight into the decomposition problem in general.

Several improvements to the selection criterion are possible, which would further focus the search. Further theoretical or statistical analysis of the performance of the algorithm is needed. The development of estimates (instead of a pessimistic upper bound) for $S_{\max}$ and S_{av} would allow a better characterization of the complexity of the decomposition. Finally, the possibilities for exploiting parallelism in the decomposition process itself should be explored.

b. Top-down versus bottom-up strategies

A complementary decomposition algorithm, proceeding in a top-down fashion, has not been developed or explored. Still, a top-down procedure would be intuitively appealing. The difficulties encountered in such an approach are different from those in a bottom-up strategy, but the exact nature of these differences remains to be explored. Insight gained from understanding a complementary strategy would undoubtedly shed additional light on the bottom-up strategy as well. A unification of both approaches may be the final goal.

c. Incompletely specified functions.

The existing decomposition algorithm does not assume that the given goal function is completely specified: it is already dealing with *don't cares* internally. However, so far we have always considered cases of completely specified functions, and possible shortcuts with *don't cares* have not been examined. Intuitively, the decomposition problem is simpler if the goal function is specified only partially. It would again be desirable to do a theoretical or statistical analysis of this issue and identify appropriate ways of treating *don't cares.*

6.2.4. Learning by Example

Our research on this topic has been only preliminary, but the following two issues appear worthwhile to explore.

a. Distributed control

The assumptions of extreme distributed control, where the nodes of the perceptron are entirely independent of each other, result in great difficulties for learning by example. In general, as in the decomposition problem, a direct assignment is not possible if only local information is available. The local controllers must be allowed to communicate with each other and exchange information.

Additionally, fan-out connections introduce nontrivial redundancy and hence the assignments are not unique. Therefore, a search mechanism is necessary in the coordination between the nodes. Further research should examine possible communication strategies that assign responsibility as each example is shown.

b. Incompletely specified functions

The case where not all examples are available is the most important issue for future research. In typical applications, an exhaustive set of examples is indeed not available and the goal is to provide a generalization. Furthermore, applications will often have *don't cares* in the examples themselves.

Both cases of *don't cares* have not been studied yet. Intuitively, they make the problem of learning by example simpler, but how this can be incorporated into a learning strategy is unclear.

BIBLIOGRAPHY

Abel77. Abelson, H., "Computational Geometry of Linear Threshold Functions," *Information and Control* **34**(1), pp. 66-92 (May 1977).

Ager82. Agerwala, T. and Arvind, "Data Flow Systems," *Computer* **15**(2), pp. 10-13 (February 1982).

Aida83. Aida, H., H. Tanaka, and T. Moto-Oka, "A Prolog Extension for Handling Negative Knowledge," *New Generation Computing* **1**(1), pp. 87-91 (1983).

Aker78. Akers, S.B., "Binary Decision Diagrams," *IEEE Transactions on Computers* **C-27**(6), pp. 509-516 (June 1978).

Alde75. Alder, M.D., "A Convergence Theorem for Hierarchies of Model Neurones," *SIAM Journal on Computing* **4**(4), pp. 491-506 (December 1975).

Alek68a. Aleksander, I. and R.C. Albrow, "Adaptive Logic Circuits," *Computer Journal* **11**(1), pp. 65-71 (May 1968).

Alek68b. Aleksander, I. and E.H. Mamdani, "Microcircuit Learning Nets: Improved Recognition by Means of Pattern Feedback," *Electronics Letters* **4**(20), pp. 425-426 (4 October 1968).

Alek70a. Aleksander, I., "Microcircuit Learning Nets: Hamming-Distance Behaviour," *Electronics Letters* **6**(5), pp. 134-136 (5 March 1970).

Alek70b. Aleksander, I., "Some Psychological Properties of Digital Learning Nets," *International Journal of Man-Machine Studies* **2**, pp. 189-212 (1970).

Alek71. Aleksander, I., "Comments on 'Universal Logic Modules and

Their Applications','' *IEEE Transactions on Computers* C-20(5), pp. 586-587 (May 1971).

Alek78. Aleksander, I., ''Structure/Function Considerations for Digital Systems that Contain Polyfunctional Elements,'' *Computers and Digital Techniques* 1(4), pp. 165-170 (October 1978).

Alek79. Aleksander, I. and T.J. Stonham, ''Guide to Pattern Recognition Using Random-Access Memories,'' *Computers and Digital Techniques* 2(1), pp. 29-40 (February 1979).

Alek83a. Aleksander, I., T.J. Stonham, and B.A. Wilkie, ''Recognition Apparatus,'' U.K. Patent Application GB 2,112,194 A (13 July 1983).

Alek83b. Aleksander, I. and P. Burnett, *Reinventing Man,* Holt, Rinehart and Winston, New York (1983).

Alek84a. Aleksander, I., ''Memory Networks for Practical Vision Systems: Design Calculations,'' pp. 197-214 in *Artificial Vision for Robots,* ed. I. Aleksander, Chapman & Hall, New York (1984).

Alek84b. Aleksander, I., ''Emergent Intelligence from Adaptive Processing Systems,'' pp. 215-233 in *Artificial Vision for Robots,* ed. I. Aleksander, Chapman & Hall, New York (1984).

Alek85. Aleksander, I., ''Intelligent Digital Systems,'' pp. 273-308 in *Advanced Digital Information Systems,* ed. I. Aleksander, Prentice/Hall International (1985).

Amar64. Amarel, S., G. Cooke, and R.O. Winder, ''Majority Gate Networks,'' *IEEE Transactions on Electronic Computers* EC-13(1), pp. 4-13 (February 1964).

Arms71. Armstrong, W.W., ''Trainable Digital Apparatus,'' United States Patent No. 3,613,084 (October 12, 1971).

Arms76. Armstrong, W.W., ''Adaptive Boolean Logic Element,'' United States Patent No. 3,934,231 (January 20, 1976).

Arms78. Armstrong, W.W. and J. Gecsei, ''Architecture of a Tree-Based Image Processor,'' *Twelfth Asilomar Conference on Circuits,* *Systems & Computers,* Pacific Grove, CA, pp. 345-349 (November 6-8, 1978).

Arms79. Armstrong, W.W. and J. Gecsei, "Adaption Algorithms for Binary Tree Networks," *IEEE Transactions on Systems, Man and Cybernetics* SMC-9(5), pp. 276-285 (May 1979).

Atla81. Atlan, H., F. Fogelman-Soulie, J. Salomon, and G. Weisbuch, "Random Boolean Networks," *Cybernetics and Systems* 12(1-2), pp. 103-121 (January-June 1981).

Bart81a. Barto, A.G., R.S. Sutton, and P.S. Brouwer, "Associative Search Network: A Reinforcement Learning Associative Memory," *Biological Cybernetics* **40** (3), pp. 201-211 (May 1981).

Bart81b. Barto, A.G. and R.S. Sutton, "Landmark Learning: An Illustration of Associative Search," *Biological Cybernetics* **42**(1), pp. 1-8 (November 1981).

Bart82a. Barto, A.G., R.S. Sutton, and C.W. Anderson, "Neuron-Like Adaptive Elements That Can Solve Difficult Learning Control Problems," Coins Technical Report 82-20, Computer and Information Science Department, University of Massachusetts, Amherst, MA (1982).

Bart82b. Barto, A.G., C.W. Anderson, and R.S. Sutton, "Synthesis of Nonlinear Control Surfaces by a Layered Associative Search Network," *Biological Cybernetics* **43**(3), pp. 175-185 (April 1982).

Bloc61. Block, H.D., "Analysis of Perceptrons," *Proceedings of the Western Joint Computer Conference,* Los Angeles, CA, pp. 281-289 (May 9-11, 1961).

Bloc62a. Block, H.D., "The Perceptron: A Model of Brain Functioning I," *Reviews of Modern Physics* **34**(1), pp. 123-135 (January 1962).

Bloc62b. Block, H.D., B.W. Knight Jr., and F. Rosenblatt, "Analysis of a Four-Layer Series-Coupled Perceptron II," *Reviews of Modern Physics* **34**(1), pp. 135-142 (January 1962).

Bloc70. Block, H.D., "A Review of 'Perceptrons: An Introduction to Computational Geometry' by M. Minsky and S. Papert, 1969," *Information and Control* **17**(5), pp. 501-522 (December 1970).

Bobr78. Bobrowski, L., "Learning Processes in Multilayer Threshold Nets," *Biological Cybernetics* **31** (1), pp. 1-6 (November 1978).

Bow84. Bow, S.-T., *Pattern Recognition*, Marcel Dekker, Inc., New York, NY (1984).

Brow85. Brownston, L., R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5*, Addision-Wesley Publishing Company, Inc. (1985).

Came60. Cameron, S.H., "An Estimate of the Complexity Requisite in a Universal Decision Network," *Bionics Symposium*, Dayton, Ohio, pp. 197-212 (13-15 September 1960).

Cern79. Cerny, E., D. Mange, and E. Sanchez, "Synthesis of Minimal Binary Decision Trees," *IEEE Transactions on Computers* **C-28**(7), pp. 472-482 (July 1979).

Cloc81. Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, Springer-Verlag (1981).

Cruz83. Cruz, C.A. and H.J. Myers, "Associative Networks II," Report 8/465-3135, IBM Palo Alto Scientific Center, Palo Alto, CA (October 7, 1983).

Curt61. Curtis, H.A., "A Generalized Tree Circuit," *Journal of the ACM* **8**(4), pp. 484-496 (October 1961).

Curt63. Curtis, H.A., "Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory," *Journal of the ACM* **10**(4), pp. 562-581 (October 1963).

Duda73. Duda, R.O. and P.E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, Inc. (1973).

Forg81. Forgy, C.L., "OPS5 User's Manual," CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (July 1981).

Frie75. Friedman, A.D. and P.R. Menon, *Theory and Design of Switching Circuits*, Computer Science Press, Inc., Woodland Hills, CA

(1975).

Fuku75. Fukushima, K., ''Cognitron: A Self-Organizing Multilayer Neural Network,'' *Biological Cybernetics* **20**(3/4), pp. 121-136 (1975).

Fuku80. Fukushima, K., ''Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position,'' *Biological Cybernetics* **36**, pp. 193-202 (1980).

Fuku84. Fukushima, K., ''A Hierarchical Neural Network Model for Associative Memory,'' *Biological Cybernetics* **50**(2), pp. 105-113 (1984).

Gall85a. Gallant, S.I., ''The Pocket Algorithm for Perceptron Learning,'' Technical Report SG-85-19, College of Computer Science, Northeastern University, Boston, MA (January 2, 1985).

Gall85b. Gallant, S.I., ''Knowledge Representation Using Linear Discriminant Networks,'' Technical Report SG-85-29, College of Computer Science, Northeastern University, Boston, MA (October 24, 1985).

Haye83. Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, ''An Overview of Expert Systems,'' pp. 3-29 in *Building Expert Systems*, ed. F. Hayes-Roth, D.A. Waterman, and D.B. Lenat, Addison-Wesley Publishing Company, Inc., Reading, MA (1983).

Hay60. Hay, J.C., F.C. Martin, and C.W. Wightman, ''The MARK I Perceptron - Design and Performance,'' *IRE International Convention Record* **2**, pp. 78-87 (1960).

Hebb49. Hebb, D.O., *The Organization of Behavior,* John Wiley & Sons, Inc., New York (1949).

Hell84. Helly, J.J. Jr., ''A Distributed Expert System for Space Shuttle Flight Control,'' Report CSD-840038, Computer Science Department, University of California, Los Angeles, CA (1984).

High73. Hight, S.L., ''Complex Disjunctive Decomposition of Incompletely Specified Boolean Functions,'' *IEEE Transactions on Computers* C-22(1), pp. 103-110 (January 1973).

Hint81. Hinton, G.E., "Implementing Semantic Networks in Parallel Hardware," pp. 161-187 in *Parallel Models of Associative Memory*, ed. G.E. Hinton and J.A. Anderson, Lawrence Erlbaum Associates, Hillsdale, NJ (1981).

Hint84. Hinton, G.E., T.J. Sejnowski, and D.H. Ackley, "Boltzmann Machines: Constraint Satisfaction Networks that Learn," Technical Report CMU-CS-84-119, Department of Computer Science, Carnegie-Mellon University (May, 1984).

Hint85. Hinton, G.E., "Learning in Parallel Networks," *BYTE* **10**(4), pp. 265-273 (April 1985).

Hopc79. Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley Publishing Company (1979).

Hopf82. Hopfield, J.J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Science of the USA* **79**, pp. 2554-2558 (April 1982).

Hube62. Hubel, D.H. and T.N. Wiesel, "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex," *The Journal of Physiology* **160**(1), pp. 106-154 (January 1962).

Jaff83. Jaffar, J., J.-L. Lassez, and J. Lloyd, "Completeness of the Negation as Failure Rule," *Proceedings of the Eight International Joint Conference on Aritificial Intelligence*, Karlsruhe, West Germany, pp. 500-506 (August 8-12, 1983).

John77. Johnson, N.L. and S. Kotz, *Urn Models and Their Application,* John Wiley & Sons (1977).

Kabl83. Kableshkov, S.O., *The Anthropocentric Approach to Computing and Reactive Machines,* John Wiley & Sons (1983).

Kauf70. Kauffman, S., "Behaviour of Randomly Constructed Genetic Nets: Binary Element Nets," pp. 18-37 in *Towards a Theoretical Biology, Vol 3: Drafts*, ed. C.H. Waddington, Edinburgh

University Press (1970).

Klop72. Klopf, A.H., "Brain Function and Adaptive Systems: A Heterostatic Theory," Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA (3 March 1972).

Klop82. Klopf, A.H., *The Hedonistic Neuron*, Hemisphere Publishing Corporation, New York (1982).

Koho82. Kohonen, T., "Self-Organized Formation of Topologically Correct Feature Maps," *Biological Cybernetics* 43(1), pp. 59-69 (January 1982).

Konh62. Konheim, A.G., "A New Class of Multilayer Series-Coupled Perceptrons," pp. 485-502 in *Self-Organizing Systems 1962*, ed. M.C. Yovits, G.T. Jacobi, and G.D. Goldstein, Spartan Books, Washington, D.C. (1962).

Kova80. Kovalevsky, V.A., *Image Pattern Recognition*, Springer-Verlag, Inc. (1980). (Originally published by Nauka, Moscow, 1977. Translated by Arthur Brown.)

Kowa79. Kowalski, R., *Logic for Problem Solving*, North-Holland (1979).

Lewi81. Lewis, H.R. and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).

Malo82. Malony, A.D., "Regular Interconnection Networks," Report No. CSD-820825, Computer Science Department, University of California, Los Angeles, CA (August 1982).

Mart83. Martinez, T., "Convergence Algorithms for Fixed Structured Logic Networks," M.S. Comprehensive Exam, University of California, Los Angeles, CA (1983).

Mart86. Martinez, T., "Adaptive Self-Organizing Logic Networks," Ph.D. Dissertation, University of California, Los Angeles, CA (1986).

Mato83. Matos, J.S. and J.V. Oldfield, "Mapping of Binary Decision Diagrams into Silicon," *1983 IEEE International Symposium on Circuits and Systems*, Newport Beach, CA 1, pp. 210-213 (May

2-4, 1983).

McCu43. McCulloch, W.S. and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* **5**(4), pp. 115-133 (December 1943).

Mich83. Michalski, R.S., J.G. Carbonell, and T.M. Mitchell, *Machine Learning,* Tioga Publishing Co, Palo Alto, CA (1983).

Mich86. Michalski, R.S., J.G. Carbonell, and T.M. Mitchell, *Machine Learning, Volume II,* Morgan Kaufmann Publishers, Inc., Los Altos, CA (1986).

Miil62. Miiller, H.S. and R.O. Winder, "Majority-Logic Synthesis by Geometric Methods," *IRE Transactions on Electronic Computers* **EC-11**(1), pp. 89-90 (February 1962).

Minn61. Minnick, R.C., "Linear-Input Logic," *IEEE Transactions on Electronic Computers* **EC-10**(1), pp. 6-16 (March 1961).

Mins69. Minsky, M. and S. Papert, *Perceptrons - An Introduction to Computational Geometry,* MIT Press, Cambridge, MA (1969).

Miya84. Miyake, S. and K. Fukushima, "A Neural Network Model for the Mechanism of Feature-Extraction," *Biological Cybernetics* **50**(5), pp. 377-384 (1984).

Moor83. Moore, D.W., "General Purpose Perceptron," Report CSD-830817, Computer Science Department, University of California, Los Angeles, CA (June 1983).

Moor85a. Moore, D.W., "Communication as a VLSI Complexity Issue," Informal Report, Computer Science Department, University of California, Los Angeles, CA (March 7, 1985).

Moor85b. Moore, D.W., "Amenable Logic Gate and Method of Testing," United States Patent No. 4,542,508 (September 17, 1985).

Moor85c. Moore, D.W. and R.A. Verstraete, "Functionally Redundant Logic Network Architectures," United States Patent No. 4,551,814 (November 5, 1985).

Moor85d. Moore, D.W. and R.A. Verstraete, "Functionally Redundant

Logic Network Architectures with Logic Selection Means," United States Patent No. 4,551,815 (November 5, 1985).

Moun80. Mountcastle, V.B., *Medical Physiology, Fourteenth Edition, Volume 1,* The C.V. Mosby Company (1980).

Muro61. Muroga, S., I. Toda, and S. Takasu, "Theory of Majority Decision Elements," *Journal of the Franklin Institute* 271(5), pp. 376-418 (May 1961).

Muro62. Muroga, S., "Generation of Self-Dual Threshold Functions and Lower Bounds of the Number of Threshold Functions and a Maximum Weight," *Proceedings of the 3rd Annual Symposium on Switching Circuit Theory and Logical Design,* Chicago, pp. 170-184 (October 7-12, 1962).

Negr64. Negrin, A.E., "Synthesis of Practical Three-Input Majority Logic Networks," *IEEE Transactions on Electronic Computers* EC-13(3), pp. 296-299 (June 1964).

Nils65. Nilsson, N.J., *Learning Machines,* McGraw-Hill, New-York (1965).

Nils80. Nilsson, N.J., *Principles of Artificial Intelligence,* Tioga Publishing Company, Palo Alto, CA (1980).

Pate76. Paterson, M.S. and L.G. Valiant, "Circuit Size is Nonlinear in Depth," *Theoretical Computer Science* 2(3), pp. 397-400 (September 1976).

Rals83. Ralston, A. and E.D. Jr. Reilly, *Encyclopedia of Computer Science and Engineering, Second Edition,* Van Nostrand Reinhold Company, New York (1983).

Rose58. Rosenblatt, F., "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review* **65** (6), pp. 386-408 (November 1958).

Rose59. Rosenblatt, F., "Two Theorems of Statistical Separability in the Perceptron," pp. 419-472 in *Mechanisation of Thought Processes,* Her Majesty's Stationary Office, London (1959).

Rose60a. Rosenblatt, F., ''Perceptron Simulation Experiments,'' *Proceedings of the IRE* 48(3), pp. 301-309 (March 1960).

Rose60b. Rosenblatt, F., ''Perceptual Generalization over Transformation Groups,'' pp. 63-100 in *Self-Organizing Systems*, ed. M.C. Yovits and S. Cameron, Pergamon Press (1960).

Rose62a. Rosenblatt, F., *Principles of Neurodynamics*, Spartan Books, Washington, D.C. (1962).

Rose62b. Rosenblatt, F., ''A Comparison of Several Perceptron Models,'' pp. 463-484 in *Self-Organizing Systems 1962*, ed. M.C. Yovits, G.T. Jacobi, and G.D. Goldstein, Spartan Books, Washington, D.C. (1962).

Rose64. Rosenblatt, F., ''A Model for Experiential Storage in Neural Networks,'' pp. 16-66 in *Computer and Information Sciences*, ed. J.T. Tou and R.H. Wilcox, Spartan Books, Inc., Washington, D.C. (1964).

Sala83. Salas, P., M. Juckler, S. Chau, R. Verstraete, D. Moore, and J. Vidal, ''UCLM Research Notes,'' Report CSD-830818, Computer Science Department, University of California, Los Angeles, CA (June 1983).

Sing62. Singleton, R.C., ''A Test for Linear Separability as Applied to Self-Organizing Machines,'' pp. 503-524 in *Self-Organizing Systems 1962*, ed. M.C. Yovits, G.T. Jacobi, and G.D. Goldstein, Spartan Books, Washington (1962).

Sriv77. Srivatsa, S.K. and N.N. Biswas, ''Karnaugh Map Analysis and Synthesis of Threshold Functions,'' *International Journal of Systems Science* 8(12), pp. 1385-1399 (December 1977).

Sriv78. Srivatsa, S.K. and N.N. Biswas, ''Karnaugh Map Synthesis of Multigate Threshold Networks,'' *International Journal of Systems Science* 9(7), pp. 785-797 (July 1978).

Ston71. Stone, H.S., ''Universal Logic Modules,'' pp. 229-254 in *Recent Developments in Switching Theory*, ed. A. Mukhopadhyay, Academic Press (1971).

Ston85. Stonham, T.J., "Practical Pattern Recognition," pp. 231-272 in *Advanced Digital Information Systems*, ed. I. Aleksander, Prentice/Hall International (1985).

Suwa84. Suwa, M., A.C. Scott, and E.H. Shortliffe, "Completeness and Consistency in a Rule-Based System," pp. 159-170 in *Rule-Based Expert Systems*, ed. B.G. Buchanan and E.H. Shortliffe, Addison-Wesley Publishing Company, Inc. (1984).

Taki78. Takiyama, R., "A General Method for Training the Committee Machine," *Pattern Recognition* **10** (4), pp. 255-259 (1978).

Taki81. Takiyama, R., "A Two-Level Committee Machine: A Representation and a Learning Procedure for General Piecewise Linear Discriminant Functions," *Pattern Recognition* **13** (3), pp. 269-274 (1981).

Tohm64. Tohma, Y., "Decomposition of Logical Functions Using Majority Decision Elements," *IEEE Transactions on Electronic Computers* **EC-13**(6), pp. 698-705 (December 1964).

Turn84. Turner, R., *Logics for Artificial Intelligence*, Ellis Horwood Limited, Chichester, UK (1984).

Uesa75. Uesaka, Y., "On the Learnability of Discriminant Functions in Pattern Recognition," *Systems-Computers-Controls* **6** (5), pp. 104-110 (September-October 1975).

Ullm69. Ullmann, J.R., "Experiments With the n-Tuple Method of Pattern Recognition," *IEEE Transactions on Computers* **C-18**(12), pp. 1135-1137 (December 1969).

Urba68. Urbano, R.H., "Structure and Function in Polyfunctional Nets," *IEEE Transactions on Computers* **C-17**(2), pp. 152-173 (February 1968).

Vali84. Valiant, L.G., "A Theory of the Learnable," *Communications of the ACM* 27(11), pp. 1134-1142 (November 1984).

Vers82. Verstraete, R., "General Purpose Perceptrons: a Boolean Treatment," M.S. Thesis, University of California, Los Angeles, CA (1982).

Vers83. Verstraete, R., ''Renewed Interest in Perceptrons,'' *Symposium IBM-NFWO*, Brussels, Belgium, pp. 57-65 (December 8, 1983).

Vida83. Vidal, J.J., ''Silicon Brains: Whither Neuromimetic Computer Architectures,'' *Proceedings of the IEEE International Conference on Computer Design - VLSI in Computers*, Port Chester, NY, pp. 17-20 (31 October-3 November 1983).

Vida85. Vidal, J.J., ''The Distributed Machine Intelligence Project,'' *The UCLA Computer Science Department Quarterly* **13**(3), pp. 143-148 (Summer 1985).

von58. von Neumann, J., *The Computer and the Brain*, Yale University Press, New Haven, CT (1958).

Wate78. Waterman, D.A. and F. Hayes-Roth, ''An Overview of Pattern-Directed Inference Systems,'' pp. 3-22 in *Pattern-Directed Inference Systems*, ed. D.A. Waterman and F. Hayes-Roth, Academic Press, New York (1978).

Widr60. Widrow, B. and M.E. Hoff, ''Adaptive Switching Circuits,'' *1960 IRE WESCON*, pp. 96-104 (August 23-26, 1960).

Widr62. Widrow, B., ''Generalization and Information Storage in Networks of ADALINE 'Neurons','' pp. 435-461 in *Self-Organizing Systems 1962*, ed. M.C. Yovits, G.T. Jacobi, and G.D. Goldstein, Spartan Books, Washington, D.C. (1962).

Widr64. Widrow, B. and F.W. Smith, ''Pattern-Recognition Control Systems,'' pp. 288-317 in *Computer and Information Sciences*, ed. J.T. Tou and R.H. Wilcox, Spartan Books, Inc., Washington, D.C. (1964).

Wils76. Wilson, M.J.D. and I. Aleksander, ''Arraylike Learning Systems,'' *Journal of Cybernetics* **6**(3-4), pp. 271-284 (July-December 1976).

Wils80. Wilson, M.J.D., ''Artificial Perception in Adaptive Arrays,'' *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-10**(1), pp. 25-32 (January 1980).

Wind63. Winder, R.O., ''Bounds on Threshold Gate Realizability,'' *IEEE*

Transactions on Electronic Computers EC-**12**(5), pp. 561-564 (October 1963).

Wind65. Winder, R.O., ''Enumeration of Seven-Argument Threshold Functions,'' *IEEE Transactions on Electronic Computers* EC-**14**(3), pp. 315-325 (June 1965).

Yaji65. Yajima, S. and T. Ibaraki, ''A Lower Bound on the Number of Threshold Functions,'' *IEEE Transactions on Electronic Computers* EC-**14**(6), pp. 926-929 (December 1965).

Yau68. Yau, S.S. and C.K. Tang, ''Universal Logic Circuits and Their Modular Realizations,'' *AFIPS Conference Proceedings*, Atlantic City, NJ **32**, pp. 297-305 (April 30-May 2 1968).

Yau70. Yau, S.S. and C.K. Tang, ''Universal Logic Modules and Their Applications,'' *IEEE Transactions on Computers* C-**19**(2), pp. 141-149 (February 1970).