Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

# A COMPARISON OF CONTINUOUS AND DISCRETE LEARNING THROUGH A GEOMETRIC REPRESENTATION

Joseph C. Pemberton

January 1989
CSD-890006

UNIVERSITY OF CALIFORNIA

Los Angeles

# A Comparison of Continuous and Discrete Learning Through a Geometric Representation

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Joseph C. Pemberton

1988

The thesis of Joseph C. Pemberton is approved.

Richard E. Korf

Richard E. Korf

David R Jefferson

David Jefferson

Jacques J. Vidal, Committee Chair

University of California, Los Angeles

1988

## Table of Contents

# Table of Figures

Page

# ACKNOWLEDGMENTS

I would like to take this opportunity to recognize the people who have contributed to this thesis, therefore, I would like to thank:

my advisor, Jacques J. Vidal, for introducing me to the field of neural networks, for many beneficial discussions and for always encouraging me to look at things in new and different ways.

Richard Korf, for giving me the time necessary to participate as a member of my thesis committee, for introducing me to artificial intelligence and for inspiring me through his dedication to academic ideals.

David Jefferson, who also gave of his time to participate as a member of my thesis committee, for always showing interest in my work and for introducing me to the field of parallel computation.

Jim Goodwin, Bruce Rosen, Jeffrey Chang, Brad Pierce, Lari Sullivan, and the rest of the DMI group, for their contributions to my working environment.

Dan Greening, for his help with mantaray and the rest of the fishnet.

my family, for always making things like this worthwhile.

and especially my wife, Tanya, without whom I could have never gotten this far.

This work was supported in part by Aerojet Electro-Systems under the Aerojet-UCLA Cooperative Research Master Agreement No. D841211, by Nasa NAG 2-302 and by an equipment grant from Hewlett Packard.

## ABSTRACT OF THE THESIS

A Comparison of Continuous and Discrete Learning

Through a Geometric Representation

by

Joseph C. Pemberton

Master of Science in Computer Science

University of California, Los Angeles, 1988

Professor Jacques J. Vidal, Chair

This thesis examines and compares a linear (activation feedback) process for adjusting the weights in a neural network to a discrete weight adjustment process. This comparison is implemented through the use of a simple perceptron and an *activation feedback-binary* node which is based on an extension to the generalized delta rule (Rumelhart *et al*, 1986). The extension allows the weight adjustment feedback signal to be different from the node's output signal. The learning rules are examined in isolation from a network. The results and observations are therefore applicable to a wide variety of network models.

The behavior of both weight adjustment processes is analyzed using an extension of the geometric interpretation of competitive learning presented by Rumelhart and Zipser (1986). In the representation, the node's weight vector is plotted onto the *weight+threshold space*, and the node's functions map to regions of the space. Weight adjustment thus corresponds to moving the node's weight vector in response to feedback, and the output function implemented by the node changes when the weight moves from one function region to another. The representation is used to compare the two weight adjustment processes in terms of their incremental and average effect on the weight vector. The results show that the linear weight adjustment process used by the *activation feedback-binary* node is superior to the discrete process in a number of respects including improved representation of the target function and immunity to noise in the target signal.

# Chapter 1
# Introduction

In the past few years, the field of artificial neural networks has become one of the fastest growing areas of research in both academia and industry. This is largely due to two somewhat independent forces. The first is a desire to better understand the functional characteristics of the human brain, and the second is driven by the demand for faster and more powerful computers. These two desires overlap in the field of artificial neural networks where brain modelers turn to computational theory for insight and computer architects turn to neural models for inspiration. This thesis is specifically concerned with comparing two fundamental procedures for adapting the structure of network interconnections. Adaptation of network interconnection strengths is recognized as an important mechanism for learning in both artificial and natural neural networks.

Neural network researchers can be classified as either neural modelers or neural architects. Neural modelers are more concerned with generating a model that best describes and predicts the function and behavior of real neurons or collections of neurons, which is a difficult task given the complexity of even the simplest neuron. They, therefore, must decide what level of behavior to explore at an early stage. Once a model has been proposed, its validity can be measured by comparing its behavior with some observed biological, psychological, or behavioral data. The model's success can also be measured by its ability to predict as yet unmeasured or unrecorded behavior of the system being modeled.

Neural architects, on the other hand, are more like neural engineers than neural scientists. In general, they are concerned with the potential computational advantages of biologically inspired models. In a sense, they rely on neurophysiology to inspire their creativity. Neural architects deal more with characterizing the functionality of their models than with determining their biological plausibility.

Of particular interest to both neural modelers and neural architects is the observation that humans can out-perform the most powerful and expensive super computers in many tasks, especially those involving sensory perception and motor coordination. Although conventional artificial intelligence (AI) research has made progress in a number of areas (most notably expert systems, advanced search algorithms and methods of symbol processing), it has shown through a lack of progress that seemingly easy tasks like vision and natural language are far from simple for standard computers and conventional programming methods. Even young children have natural language skills that far exceed the best capabilities of conventional AI approaches. This has lead many AI researchers to consider computational models based on the most successful processor, namely the nervous system.

Neural network research itself can be broken down into high level and low level models. High level modelers are interested in the behavior and functional characteristics of large collections of interconnected units. Low level modelers, on the other hand, are interested in the behavior and functional characteristics of individual units or a small collection of units. Ultimately, the high level results and observations can be described in terms of the low level models.

This thesis deals with low-level neural architecture research. Although the biological consequences and high level network implications of this work are of interest, they are not within the scope of this work and, therefore, will not be covered. Instead, this thesis will concentrate on an in-depth comparison of a linear network node and the simple perceptron, which are two common neural network building blocks. The observations and results will be shown to be applicable to a wide variety of neural models.

The first three chapters of this thesis are intended to provide background material for the reader. The basic elements and functions pertaining to neural network nodes are discussed along with some historical background. Also included is a section on notation used throughout this thesis. An extension to the generalized delta rule and the *activation feedback-binary* network node are presented in section 3.3.

This extended learning rule is illustrated and analyzed through the use of the *activation feedback-binary* network node in combination with a geometric representation which provides a framework for observing and describing the dynamic behavior of a network node. Through this model, a linear (continuous feedback) version of the generalized delta rule is compared with the simple perceptron learning procedure, and the results and observations are summarized along with suggestions for future work.

Note that a glossary of neural network terminology is provided in appendix A. It contains commonly used neural jargon as well as terms specifically related to this work. Its purpose is to assist the reader with unfamiliar terms.

# Chapter 2
# Foundations

This chapter, which is intended to provide background material for the reader, presents a general overview of neural networks and a brief history of neural network building block design. Also included is a description of the notation used to describe network signals throughout the remainder of this thesis.

## 2.1 What is a Neural Network?

The term *neural network* has been used to describe a large class of computational models that are loosely based on models of the brain. As mentioned in the introduction, some researchers are interested in the brain modeling aspects whereas others are interested in the computational advantages. These networks have also been called *artificial neural networks* to emphasize the fact that they are not strict implementations of the brain's physiology. Other names associated with neural networks include connectionist models, parallel distributed processing models and neuromorphic systems.

Regardless of the name used, neural networks typically consist of a large number of highly interconnected, simple functional units (or nodes). Networks learn by adjusting (or adapting) their network interconnection strengths in response to environmental stimuli. At least in principle, this process corresponds to current ideas on brain functionality. It is believed that processing is distributed throughout the brain, that memory exists as patterns of activations and interconnections, and that learning involves making changes to the interconnections between neurons. From a computational standpoint, neural networks distribute and intersperse processing and memory thereby circumventing the problems associated with the classical von Neumann computer architecture, such as memory and input/output bottlenecks.

The operation of a neural network, as shown in figure 2.1, can be divided into two distinct modes. The *execution* mode views the network as a set of parallel, asynchronous processing units. Environmental patterns presented to the network inputs are processed by the input units and sent to other network units (or output directly to the environment). The signals flow from unit to unit and eventually reach units that output to the environment. Clearly the processing of a network depends on the processing of the individual units and the way in which the units are interconnected. However, in the *learning* mode, the internal parameters of the network (*i.e.* the strength of the weights that connect the functional units) are changed in response to internal feedback or feedback from the environment.

Neural network models differ in the choice of functional units, the way the units are connected together, and the methods used for updating the interconnections. Network nodes will be discussed further in chapter 3. A typical network structure and sample method used for adapting network interconnections will be briefly described next.

It is assumed in this thesis that the interconnection weights are internal to the node. This means that the network configuration is simply a description of how the

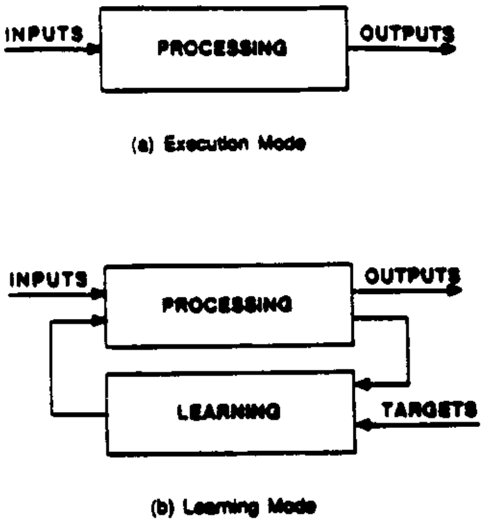


Figure 2.1: Block Diagram of Neural Network Operation

nodes are connected to each other and to the environment. One common configuration is the layered network (see figure 2.2). In this network, each node in the input layer only receives input signals from the environment and all other nodes only receive input signals from nodes in the previous layer. Nodes which don't directly communicate with the environment are called hidden nodes.

Layered networks have two advantages. The first is that the layered structure makes the network function easier to understand. Network processing can be viewed as a wave of data flowing from the input layer, through the intermediate layers, to the output layer. The second advantage is that layered structures support relatively simple methods for updating interconnections (*e.g.* backward error propagation and competitive learning).

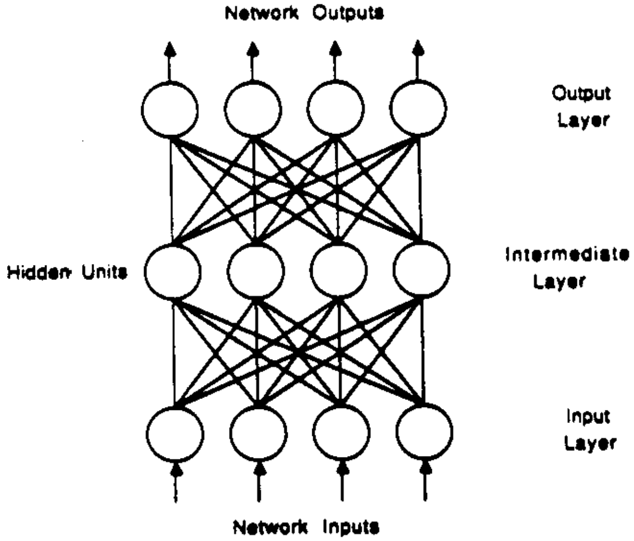


Figure 2.2: A Three Layer Network

Neural network learning involves adjusting the interconnection weights in response to input and target signals. This can be broken down into node learning rules, which describe how to adjust the node's input weights in response to input and target signals, and target generation schemes, which provide the target signals to nodes which don't receive target signals from an external agent (*e.g.* a teacher or the environment). Node learning rules will be further discussed in chapter 3.

## 2.2 Historical Background

The *perceptron* was one of the earliest network building blocks to be proposed (Rosenblatt, 1961). It is also a widely accepted name for a general class of devices that use threshold logic units (TLU's) and the rules which govern their behavior. Since its introduction in the late 1950's, there have been a number of extensions to the original design including a linear version and a more general building block description. The original perceptron was not designed to be a detailed model of a neuron, but instead it was developed to examine the behavior of networks made up of a number of perceptrons (Rumelhart and Zipser, 1986). In a sense, Rosenblatt wanted to determine if the behavioral characteristics of natural systems could be generated by combining the individual behaviors of a collection of simple functional units. Consequently, much of his work centered around characterizing the behavior of the perceptron and comparing it to physiological data.

The threshold logic unit (TLU) forms the heart of the perceptron and many other network models. It can be viewed as a pattern classifier and/or pattern recognition system. Also, it can be thought of as an adaptive logic device that makes a membership decision on the current input pattern. Both views are valid and have no effect on the underlying nature of the perceptron. From a functional perspective, the output of a TLU is a function of the inputs and its internal parameters.

As illustrated in figure 2.3, the processing components of a TLU consist of input weights, a summation unit and a threshold function unit. Input patterns are first multiplied by their associated input weights, and the products are summed to generate an activation signal. The threshold function then compares the current threshold and activation to determine a binary output signal.

A TLU can be used to implement a subset of the possible logic functions called the linearly separable functions. The family of linearly separable functions will be discussed further in chapter 3. As an example of a TLU function, consider the *and* function of two inputs. It can be implemented by setting both input weights equal to 0.5 and the threshold also equal to 0.5. The resulting threshold function is shown in table 2.1.

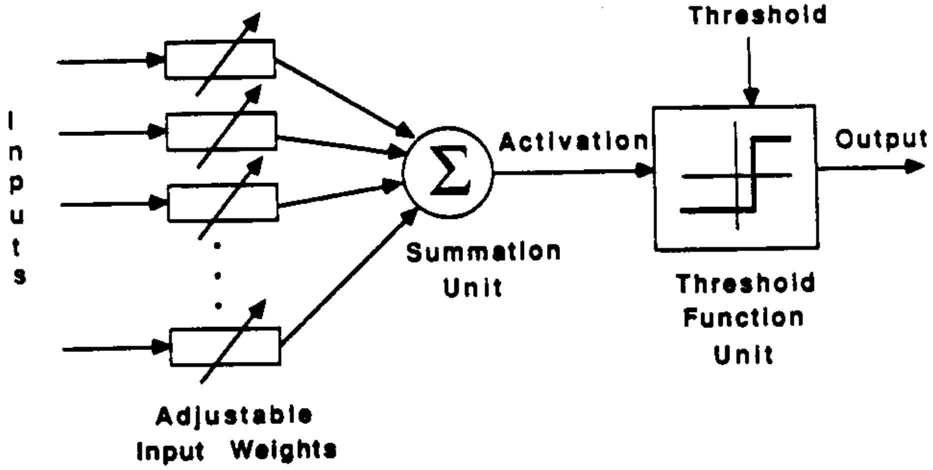


Figure 2.3: Processing Components of a Threshold Logic Unit (TLU)

| inputs | weighted sum | > threshold (0.5) ? | output |
|---|---|---|---|
| (−1, −1) | $\frac{1}{2}(-1) + \frac{1}{2}(-1) = -1$ | no | −1 |
| (−1, +1) | $\frac{1}{2}(-1) + \frac{1}{2}(+1) = 0$ | no | −1 |
| (+1, −1) | $\frac{1}{2}(+1) + \frac{1}{2}(-1) = 0$ | no | −1 |
| (+1, +1) | $\frac{1}{2}(+1) + \frac{1}{2}(+1) = +1$ | yes | +1 |

Table 2.1 Example TLU Function Implementation (*AND*)

Minsky and Papert (1969), whose work was largely confined to a one-layer simple perceptron, presented a formal yet critical analysis of the perceptron's computational power. They argued that the perceptron suffers from the same scaling problems (*e.g.* only works well for small problem domains) as serial methods of computation and is unable to identify some relatively simple yet important functions of the inputs (*e.g. XOR* and *EQUIVALENCE*). Their sharp criticism had a very negative effect on perceptron related research. However, recently Minsky has reconsidered his position, stating that the perceptron is a powerful pattern recognizer given its design simplicity (Rumelhart and Zipser, 1985).

In 1960, Widrow and Hoff presented an adaptive linear element (*adaline*) as an implementation of an algorithm for solving sets of linear equations (Widrow and Hoff, 1960). The main difference between this model and the simple perceptron is in the threshold function; the perceptron uses a discrete (binary) threshold function whereas the adaline uses a continuous (linear) threshold. The weights used in this model are updated through the Widrow-Hoff rule (later referred to as a linear version of the generalized delta rule (Rumelhart *et al*, 1986)). The characteristics of this learning rule will be explored in detail in chapter 4.

More recently, Rumelhart *et al* (1986) proposed the generalized delta rule as a general description of learning in neural network building blocks (nodes). Its form is very similar to the simple perceptron learning procedure (a method of adjusting the weights of a TLU), but the details of the model (*e.g.* output function, learning rule, etc.) can vary depending on the node's design. The simple perceptron learning rule and the Widrow-Hoff rule are special cases of this learning rule.

This thesis presents a more flexible specification of network node design than currently used with the generalized delta rule. A node based on this specification is then used to compare the linear learning procedure (as used by Widrow and Hoff) to the binary learning procedure (as used in the simple perceptron model).

## 2.3 Network Signal Notation

Neural network research papers contain a variety of signal notations. This section describes the signal notation format which will be used throughout this work. It has been designed with the following goals in mind: to be as consistent as possible with the majority of neural network related research, to be consistent with other scientific and engineering fields, and to provide a notation which is easily understandable and adaptive for future applications. This signal naming format is proposed as a standard signal notation for neural network research.

Figure 2.4 illustrates the notation used in this thesis to refer to internal signals of the perceptron and other neural network nodes. Node input and output signals are denoted by an "$x$" and have subscripts that consist of two letters and an arrow that describe the direction of signal flow. For example, $x_{i \rightarrow j}$ means the output of the $i^{th}$ signal source that is connected to an input of the $j^{th}$ node. A signal source is either a network input or node. Note that the input vector for the $j^{th}$ node is $\mathbf{x}_{\rightarrow j}$ (bold denotes a vector) and the output of the $j^{th}$ node is $x_{j \rightarrow}$. These signals can be indexed by time (*e.g.* $\mathbf{x}_j(t)$) or by input pattern (*e.g.* $\mathbf{x}_j(p)$ where $p$ is the input pattern number).

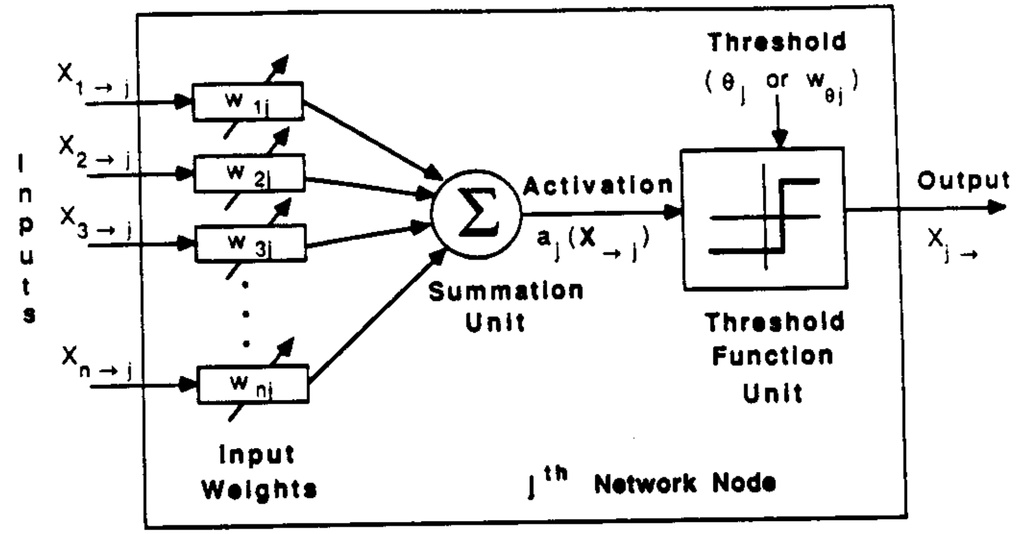


Figure 2.4: Notation for Network Node Signals

Input weights are named in a similar fashion. A particular weight of the $j^{th}$ node is referred to as $w_{ij}$ where $i$ is the source of the input signal. Note that when two subscripts are used, the order is always *source* followed by *destination*. The entire weight vector of node $j$ is expressed as $\mathbf{w}_j$ (without any source subscript). The activation (the output of the summation unit) of a node is identified by "$a_j$" where $j$ denotes the node. The target or desired activation is referenced in a similar way with the addition of a "*" superscript to indicate that it is an ideal or target value. Note that the "*" can be used to indicate the ideal or target value of any node variable (*e.g.* $a_j^*, x_{j\to}^*$). $t_j^*$ is used to refer to a generic target signal. The threshold of the $j^{th}$ node is either denoted by $\theta_j$ or $w_{\theta j}$ depending on the implementation. $\xi_j$ (or $\delta_j$) is used to denote the error signal of the $j^{th}$ node and is equal to the difference between a target signal ($t_j^{\bullet}$) and the corresponding feedback signal.

This notation can be used to describe the TLU implementation example discussed earlier. The inputs to the $j^{th}$ TLU from source $i$ and $k$ are $x_{i\to j}$ and $x_{k\to j}$, the input weights are $w_{ij}$ and $w_{kj}$, and the output signal is $x_{j\to}$. These are summarized in table 2.2 which is equivalent to table 2.1.

| $(x_{i\to j}, x_{k\to j})$ | $w_{ij}(x_{i\to j}) + w_{kj}(x_{k\to j}) = a_j$ | $> \theta_j$ (0.5) ? | $x_{j\to}$ |
|---|---|---|---|
| (−1, −1) | $\frac{1}{2}(-1) + \frac{1}{2}(-1) = -1$ | no | −1 |
| (−1, +1) | $\frac{1}{2}(-1) + \frac{1}{2}(+1) = 0$ | no | −1 |
| (+1, −1) | $\frac{1}{2}(+1) + \frac{1}{2}(-1) = 0$ | no | −1 |
| (+1, +1) | $\frac{1}{2}(+1) + \frac{1}{2}(+1) = +1$ | yes | +1 |

Table 2.2 Example TLU Function Implementation Using Signal Notation

The target signal ($t_j^{\bullet}$) can be used by the TLU to update its input weights and threshold. This obviates the need to preset the weights and threshold to the correct function. Autonomous adjustment of the TLU weights and threshold will be further discussed in the next two chapters.

# Chapter 3
# Building Blocks for Neural Networks

The design of a neural network consists of specifying the functionality of the nodes, their configuration (*e.g.* the number of nodes, the number of layers, which nodes are input, hidden or output nodes, how the nodes are connected together, etc.), and a method of generating internal target signals. A network building block is defined to be a node and its associated input weights. This chapter presents a general description of a neural network node along with a detailed description of the simple perceptron which serves as an example node. Also presented is the *activation feedback-binary* node which is based on an extension to the generalized delta rule node model.

## 3.1 Neural Network Nodes

A network node's functional description must include a discussion of its input and output signal format as well as its logical operation. The logical operation of a node includes two distinct subcomponents; the *processing function* governs how a node transforms input patterns into output signals, and the *learning rule* determines how the internal parameters of a node should change in response to the target signal.

### 3.1.1 Signal Format

The formats of signals entering and leaving a node vary depending on the particular network model. In the classical perceptron model, for example, the value of inter-node signals is restricted to the set {0,1}, whereas the inter-node signals in the *adaline* model can take on any real value. Note that although the input and output signal formats can be specified independently, they are typically the same to allow for cascading the output of one unit to the input of another.

There are three signal format options. The first determines whether the signal is discrete or continuous. A discrete signal can take on values that are members of a specific set, such as binary signals which are restricted to the set {0,1}. A continuous signal, on the other hand, can take on any real value.

The second format choice indicates whether the signal is symmetric or asymmetric. For example, a discrete signal is symmetric if for every positive member of the set of discrete values, there is a corresponding negative member of equal magnitude. A symmetric binary signal takes on values from the set $\{-1, +1\}$.

The final option determines whether the signal is bounded or unbounded. If a signal is bounded, its value falls within a prespecified range (*e.g.* $c_1 \le x \le c_2$, where $c_1$ and $c_2$ are constants). An unbounded signal may take on arbitrarily large values. Note that binary signals are by definition bounded. A summary of the signal format choices is presented in table 3.1.

| Signal Format | | | Example ($x$) |
|---|---|---|---|
| discrete | symmetric | bounded | $x \in \{-1, +1\}$ |
| discrete | symmetric | unbounded | $x \in \{..., -2, -1, +1, +2, ...\}$ |
| discrete | asymmetric | bounded | $x \in \{0, 1\}$ |
| discrete | asymmetric | unbounded | $x \in \{0, 1, 2, ...\}$ |
| continuous | symmetric | bounded | $-c \le x \le +c$ |
| continuous | symmetric | unbounded | $-\infty \le x \le +\infty$ |
| continuous | asymmetric | bounded | $-c_1 \le x \le +c_2$ |
| continuous | asymmetric | unbounded | $0 \le x \le +\infty$ |

Table 3.1 Summary of Node Signal Formats

### 3.1.2 Node Processing

The node processing (or output) function determines how the node transforms input patterns into output signals. In the most general interpretation of a neural network, the output can be an arbitrary function; however, if the processing is based on a threshold logic unit node (TLU), the output is typically a function of the input pattern, current weights and threshold. The weights and threshold thus completely determine the mapping between input patterns and output signals.

Although the output of a TLU is determined by the inputs, weights, and threshold, it is convenient to express the output as a function of the threshold and an internal state variable (*i.e.* the activation). In general, the output function of the $j^{th}$ node is

$$x_{j\rightarrow}(t) = f(\theta_j(t), a_j(t))$$

where $f()$ is the threshold function, $\theta_j(t)$ is the threshold of the *jth* node, and $a_j(t)$ is the node activation at time $t$. The activation in turn is a function of the inputs and input weights:

$$a_j(t) = g(\mathbf{x}_{\rightarrow j}(t), \mathbf{w}_j(t)) = \sum_{i=1}^{n} w_{ij}(t)\, x_{i\rightarrow j}(t) \qquad (3.1)$$

where $n$ is the number of inputs to node $j$.

The threshold function provides a mapping between the node's activation and its output signal. Discrete, linear and sigmoidal threshold functions are shown in figure 3.1. Not shown is a stochastic threshold function which is used in some neural models. For the stochastic function, the output is binary and depends on a probability distribution.

Other processing models have been proposed. One example is the *sigma-pi* model (Rumelhart *et al*, 1986) which applies a threshold function to a combination of a weighted sums (sigma) and products (pi) of the inputs.

A TLU can only implement functions that are linearly separable. The term *linearly separable* has been defined in a number of texts on pattern recognition (*e.g.* Duda and Hart, 1970 and Nilsson, 1965). Briefly, a function which divides the set of

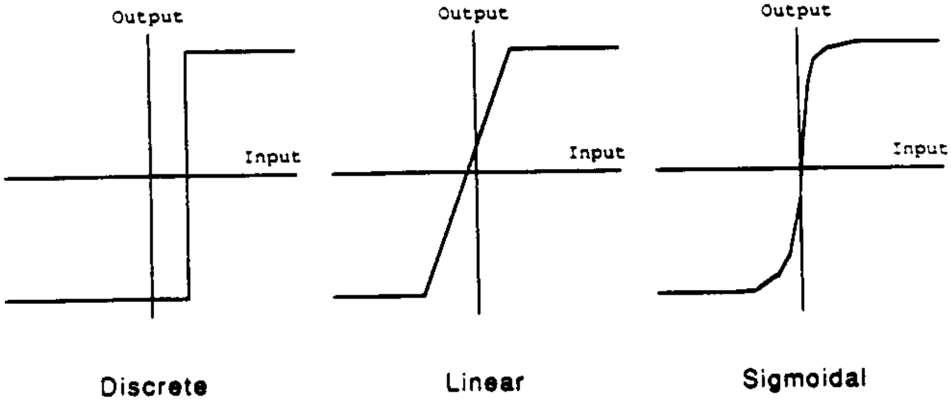


Figure 3.1: Three Threshold Functions

possible inputs into two subsets is linearly separable if and only if a linear function $g$ exists such that:

$$g(\mathbf{x}_{\rightarrow j}) \geq 0\,;\ \mathit{when}\, x_{j\rightarrow} = +1$$

$$g(\mathbf{x}_{\rightarrow j}) < 0\,;\ \mathit{when}\, x_{j\rightarrow} = -1$$

Alternatively, a function is linearly separable if and only if it can be implemented as a separation of the $n$ dimensional input space into two regions (one for $x_{j\rightarrow} = +1$ and one for $x_{j\rightarrow} = -1$) by an $n-1$ dimensional hyperplane.

Although there is no closed formula for the number of functions that can be implemented by an $n$ input TLU (Verstraete, 1982), the upper bound on the number of linearly separable functions of $n$ inputs is $2^{n^2}$ (Winder, 1965). This is significantly less that the total number of possible logic functions of $n$ inputs $(2^{2^n})$, and means that the computational power of a single TLU is limited for a large number of inputs. Fortunately, this limitation can be overcome by connecting a collection of TLU's together into a network (*e.g.* the *committee machine* (Nilsson, 1965)).

For example, *XOR* of two inputs is not a linearly separable function and therefore cannot be implemented by a single TLU. It can, however be implemented by three TLU's as demonstrated in figure 3.2.

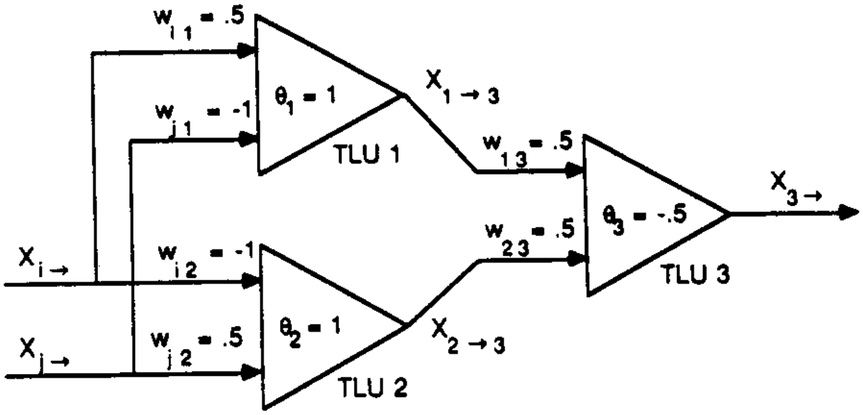


Figure 3.2: TLU Implementation of XOR

In summary, the node's output can be any function of the input signals and any internal state variables. However, if the node is implemented with a TLU, then the processing design options are reduced to specifying the threshold function. A single TLU is only able to implement functions that are linearly separable. Functions which are not linearly separable can be implemented by a network of TLU's.

### 3.1.3 Node Adaptation (Learning)

A learning rule describes how the internal structure of a node should be changed in response to a target signal. It is assumed in this thesis that the target signal has been provided; therefore all analysis of learning rules will be done without regard to the target signal source. In this way the analysis of learning procedures is applicable to a wide range of neural network models. The objective of the learning rule is to change the node's internal parameters (*e.g.* the weights and threshold) so as to improve some measure (*e.g.* mean squared difference between output and target) of its performance on future input patterns.

Since the node's processing function can be arbitrary, the learning rule associated with the processing function can also be chosen arbitrarily. The learning rule for a TLU specifies how the input weights and threshold should be modified in response to the target signal so that either its activation or output signal matches a target signal as closely as possible.

A number of rules for adjusting the input weights of a TLU have been proposed and studied. Most, if not all, of these rules are descended from the *Hebbian* learning rule. Hebb (1949) was concerned with the firing of neurons; he suggested that if a (presynaptic) neuron fires and shortly thereafter the postsynaptic neuron fires (possibly as a consequence of the presynaptic signal), then the path connecting the two neurons should be strengthened (*i.e.* the weight should be increased). A more general interpretation of this rule (Rumelhart *et al*, 1986) is

$$\Delta w_{ij} = g(x_{j\rightarrow}(t), t_j^*(t))\, f(x_{i\rightarrow j}(t), w_{ij})$$

Basically, the change to the input weights is described as a product of two functions, one of which depends on the output and target signals (g) and another which depends on the inputs and input weights (f).

Many variations on this rule have been studied, for example Grossberg's model (Rumelhart *et al*, 1986) uses the following equation for adjusting the input weights:

$$\Delta w_{ij}(t) = \eta[x_{i \to j}(t) - w_{ij}(t)]\, t_j^*(t)$$

where $\eta$ is the learning rate constant (or gain). In this model, the weight vector is seen as an approximation of the input vector. Here, the target signal is generated using a competitive learning scheme. The target signal to the unit whose weight vector is the best approximation of the input vector is set to one whereas the other target signals are set to zero within a competing region. Each unit thus learns a set of weights that best represent a set of input vectors.

The analysis that will be presented in chapter 4 is based on another variation of the Hebbian learning rule which has been called the Widrow-Hoff rule (Sutton and Barto, 1981).

$$\Delta w_{ij}(t) = \eta(a_j^*(t) - a_j(t))\, x_{i \to j}(t)$$

where $a_j^*(t)$ is the target activation signal and $a_j(t)$ is the actual activation signal (in this model the activation is the output signal). This learning rule was first developed as an iterative method for solving a set of linear difference equations. It is also referred to as the ***delta rule*** (Rumelhart *et al*, 1986) because the amount of weight adjustment is proportional to the difference between the target activation and the actual activation.

The *generalized delta rule* (Rumelhart *et al*, 1985), a further refinement on Hebbian learning, provides a general framework for specifying the way that the input weights of a TLU should change in response to a target signal. In its general form, the change to the weight connecting the output of the $i^{th}$ signal source to an input of the $j^{th}$ node is:

$$\Delta w_{ij}(t) = \eta(t_j^*(t) - x_{j\rightarrow}(t))\, x_{i\rightarrow j}(t) \tag{3.2}$$

where $t_j^*(t)$ is a target signal for the $j^{th}$ node, $x_{j\rightarrow}(t)$ is the actual output of the node and $x_{i\rightarrow j}(t)$ is the value of the input to node $j$ from the $i^{th}$ signal source. A block diagram of the generalized delta rule weight adjustment process is shown in figure 3.3. The Widrow-Hoff rule is just a special case of the generalized delta rule where

$$t_j^*(t) = a_j^*(t); \quad x_{j\rightarrow}(t) = a_j(t).$$

Notice that the weight adjustment rule as specified in equation 3.2 is completely determined by the target, input and output signals and the learning rate constant (η). Since the target and input signals are provided by some external source, the only controllable parameter is the node's output which in turn is specified by the threshold function. This means that once the node's input and target signal format and threshold function have been chosen, the learning rule is uniquely defined within the generalized delta rule framework. An extension which removes this restriction on node design will be presented in section 3.3.

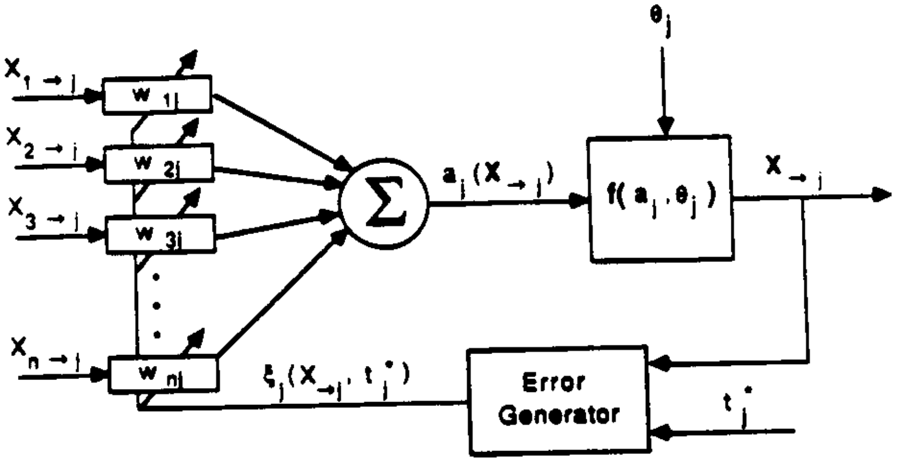


Figure 3.3: Network Node Based on the Generalized Delta Rule

## 3.2 The Simple Perceptron

Perhaps the best known variation on Hebbian learning is the simple perceptron learning rule. A block diagram of the simple perceptron is shown in figure 3.4. The input, output and target signals are binary. For this discussion the binary signals were chosen to be symmetric so that the weights can potentially be updated for both input values. A symmetric representation has been used in other models for the same reason as well as to remove the inherent asymmetry of zero-one valued outputs (Hampson and Volper, 1987). The signal format can be summarized as:

$$x_{i\to j}(t),\ x_{j\to}(t),\ t_j^*(t) \ \in\ (-1, +1)$$

The processing components of a simple perceptron are the same as that of a threshold logic unit (see figure 2.2). This means that there is a weight associated with

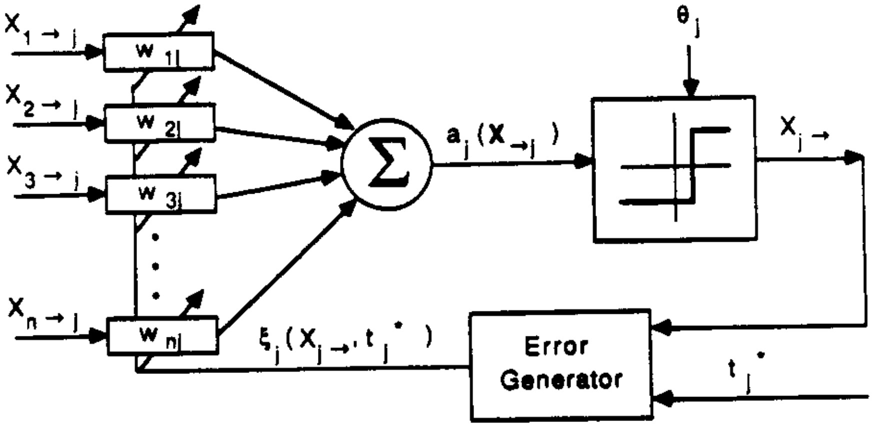


Figure 3.4: Block Diagram of the Simple Perceptron

each input, a summation unit, a threshold and a threshold function unit. Because the output signal format is binary, the threshold function is necessarily discrete. Note that the transition point for the threshold function is determined by the threshold.

The output (threshold) function for the simple perceptron is:

$$x_{j\to}(t) = \begin{cases} +1; \text{ if } a_j(t) > \theta_j(t) \text{ (or } (a_j(t) - \theta_j(t)) > 0) \\ -1; \text{ if } a_j(t) \le \theta_j(t) \text{ (or } (a_j(t) - \theta_j(t)) \le 0) \end{cases} \tag{3.3}$$

where $\theta_j(t)$ is the threshold value of the $j^{th}$ node. Notice that comparing the activation to the threshold is equivalent to comparing the difference between the activation and the threshold to zero. This means that the output can be equivalently defined in terms of the effective activation:

$$x_{j\rightarrow}(t) = +1;\ \text{if}\ a_j^{eff}(t) > 0$$

$$-1;\ \text{if}\ a_j^{eff}(t) \le 0$$

where the effective activation is:

$$a_j^{eff}(t) = a_j(t) - \theta_j(t) = \sum_{i=1}^{n} [w_{ij}\, x_{i\rightarrow j}(t)] - \theta_j(t)$$

and $n$ is the number of inputs.

Note that the threshold has the effect of ***shifting*** the transition point of the threshold function. There are two other ways to accomplish this shift of the activation without effecting the node's functionality. One is to subtract the threshold from the activation to generate an effective activation which is then compared to an unbiased threshold function (figure 3.5a). The other is to treat the threshold as a weight whose input is tied to −1 and add it in with the other inputs (figure 3.5b). The latter method is useful because it emphasizes the fact that the threshold ($\theta_j$) can be learned as a weight ($w_{\theta j}$).

The simple perceptron's output can be expressed in terms of the inputs, weights, and threshold by substituting equation 3.1 for the activation in equation 3.3 to yield:

$$x_{j\rightarrow}(t) = +1;\ \text{if}\ \sum_{i=1}^{n} w_{ij}\, x_{i\rightarrow j}(t) > \theta_j(t) \tag{3.4}$$

$$-1;\ \text{if}\ \sum_{i=1}^{n} w_{ij}\, x_{i\rightarrow j}(t) \le \theta_j(t)$$

Notice that equation 3.4 separates the input space into two regions. One region contains those input patterns that generate an activation which is greater than the threshold. The other region contains the remaining input patterns whose resulting

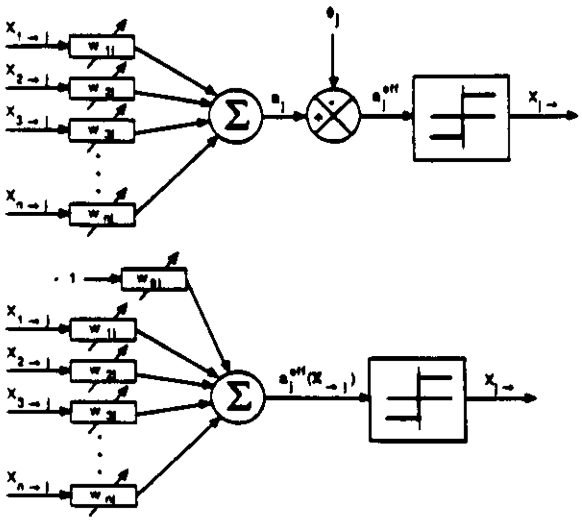


Figure 3.5: Two Alternate Implementations of the Threshold

activation is equal to or less than the threshold. A network node that uses a discrete threshold function is thus a discrete pattern classifier.

The simple perceptron learning rule describes the change to weight $w_{ij}$ as

$$\Delta w_{ij}(t) = \eta \, (x^*_{j\to}(t) - x_{j\to}(t)) \, x_{i\to j}(t) \tag{3.5}$$

where $\eta$ is a learning rate constant. $x_{j\to}(t)$ has been defined as the output of the $j^{th}$ node and $x^*_{j\to}(t)$ is the target (or desired) output signal provided by some agent external to the perceptron ($x_j^*(t) = t_j^*(t)$). An asymmetric binary input signal (*i.e.* $\in \{0,1\}$) would result in no change to the weights half of the time (*i.e.* whenever the input is zero). For this reason a symmetric binary signal format was chosen here.

Close examination of equation 3.5 leads to two interesting observations. The first is that the weights are only updated when the output and the target signals don't match. This means that the perceptron's output must be incorrect before it can learn. The second observation is that when the input weights are updated the amount of adjustment is a discrete step of either $+2\eta$ or $-2\eta$:

$$\Delta w_{ij}(t) = \eta \left( (\pm 1) - (\pm 1) \right) (\pm 1)$$

$$= (\pm 2\eta \; or \; 0)$$

In summary, the node described here, which is based on the simple perceptron, is a threshold logic unit with symmetric, binary input, output and target signals. It uses the simple perceptron learning rule that only updates the weights when the output differs from the target signal. When the weights are updated, their value is either increased or decreased by a discrete step equal in magnitude to twice the learning rate. For the remainder of this thesis, this node will be referred to as the simple perceptron and will be used for comparison to a new node which is presented in the next section.

### 3.3 A More Flexible Node Model

As was discussed in section 3.1, the only controllable parameter in the design of a node based on the generalized delta rule is the node's threshold function. Consequently, the format of the node's output and learning rule are determined by the same parameter (the threshold function). This section presents an extension to the generalized delta rule node that allows the format of the output function and learning rule to be specified independently. An example that demonstrates this extension is an *activation feedback-binary* node which has a continuous learning rule and a discrete output function. This new node makes it possible to compare the simple perceptron's discrete learning rule to a continuous feedback learning rule (see chapter 6).

### 3.3.1 A General Extension

A general node based on an extension to the generalized delta rule framework is shown in figure 3.6. Note that this model is more flexible because it allows the output function and learning rule to be specified separately.

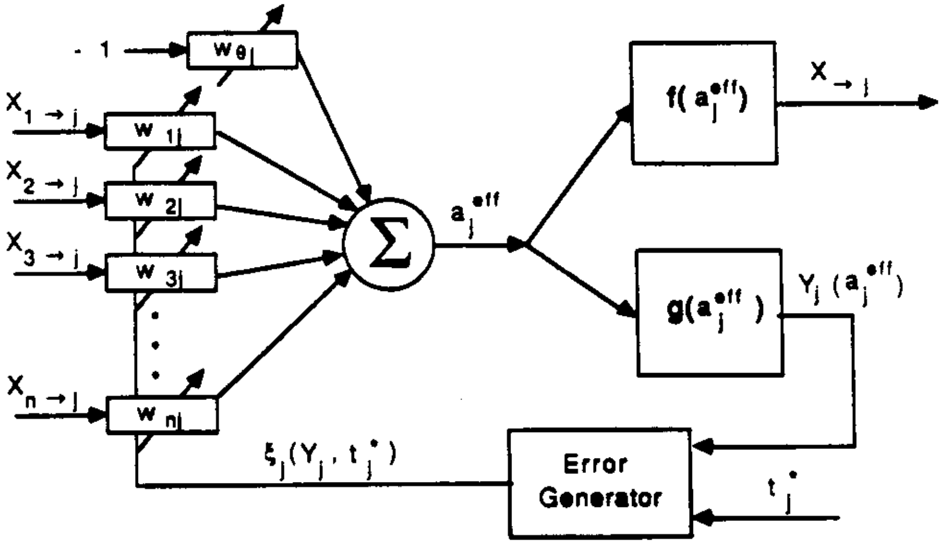


Figure 3.6: A Node Based on an Extension to the Generalized Delta Rule

The learning rule for this extension is:

$$\Delta w_{ij}(t) = \eta(t_j^*(t) - y_j(t))\, x_{i\to j}(t) \qquad (3.6)$$

where $y_j(t)$ is some function of the inputs, the input weights and perhaps some internal state variables, but it is not necessarily equal to the node's output ($x_{j\to}(t)$). The choice for a node's output and weight update functions can now be made independently. Of course, $y_j(t)$ can be identical to the node's output if desired, but this is up to the node designer and is no longer a restriction of the model.

This extension adds flexibility by allowing a node to have a binary output but use a linear weight adjustment process, as presented below.

### 3.3.2 Activation Feedback-Binary (AFB) Network Node

An *activation feedback-binary* network node can be described as a hybrid node that has the output function of a simple perceptron and the learning rule of a linear node. The node is called "*activation feedback*" because it uses *activation feedback* in the weight adjustment process. It is also called "*binary*" because the output is a discrete (binary) function of the activation.

Since the *AFB* node and the simple perceptron have identical output functions, the only difference between them is the method used to update their weights. This makes it possible to compare discrete and linear learning rules through a comparison of the behavior of these nodes.

The simple perceptron learning rule is a special case of this extension to the generalized delta rule where the feedback signal is the output of a discrete threshold function which is identical to the node's output function ($y_j(t) = x_{j\rightarrow}(t)$). The weight adjustment in an *AFB* network node is accomplished using the extension described by equation 3.6.

$$\Delta w_{ij}(t) = \eta\,(t_j^*(t) - a_j(t))\,x_{i\rightarrow j}(t) \qquad (3.7)$$

where $y_j(t)$ has been replaced by the node's activation ($a_j(t)$). Note that this learning rule is the same as the Widrow-Hoff rule; however, the output function of this node is discrete rather than continuous. This *AFB* node is shown in figure 3.7. Note that the feedback threshold function ($g$ ()) has been replaced by a direct connection.

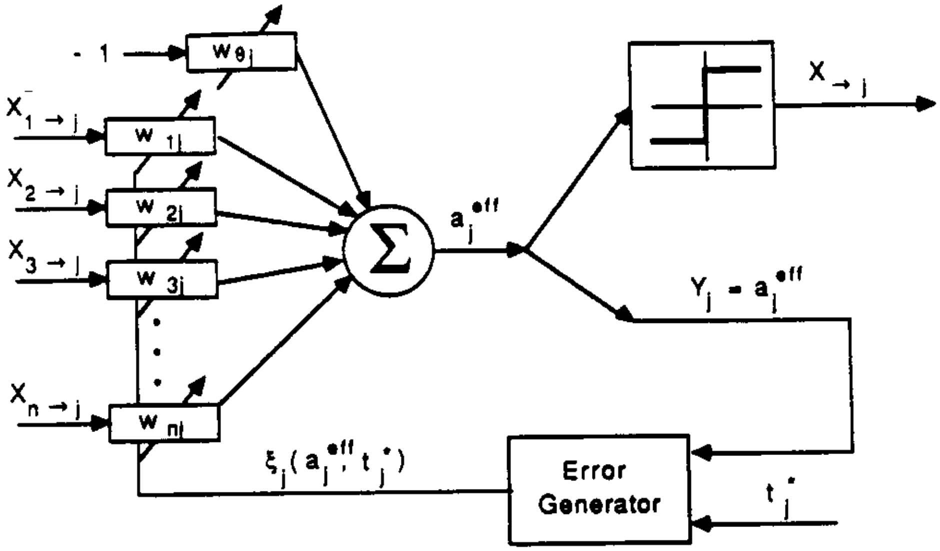


Figure 3.7: Activation Feedback Binary (AFB) Network Node

The change from output feedback (simple perceptron) to activation feedback (*AFB* node) has a dramatic effect on the weight adjustment process. One consequence of this change is that the weights are updated by an amount which is proportional to the difference between the target and the current activation. A larger difference causes a larger weight change. Note that the weights are updated even if the output is correct, as long as the current activation differs from the target. This is in contrast with the simple perceptron learning rule where the weights are updated by discrete steps and the update only occurs when the output is incorrect.

A second effect of changing to a linear weight adjustment process is that, when the input patterns are presented fairly (*i.e.* a uniform input pattern distribution), the iterative weight adjustment process can be reduced to a set of independent linear difference equations. This will be examined in the next chapter.

# Chapter 4
# A Mathematical Analysis of Linear Weight Adjustment

This chapter presents a mathematical analysis of the linear (activation feedback) weight adjustment process. The results presented here will lay the final groundwork for the geometric representation to be discussed in chapter 5. It will be shown that an iterative weight adjustment process can be simplified for a known training set and that for some training sets, the weight adjustment process reduces to a set of linearly independent equations. The weight update equations will then be further reduced to determine how the steady state weight vector depends on the training set. A two input node is used to simplify the discussion, but the results are extended to nodes with an arbitrary number of inputs.

## 4.1 Simplification of the Weight Adjustment Process

This section presents a simplification of the linear weight adjustment process. It is assumed here that the learning mode consists of the following discrete sequential steps: input pattern and target signal presentation, output generation, error calculation, and synchronous weight adjustment. This assumption simplifies the analysis and makes it possible to analyze each step of the weight adjustment process separately. It should be noted that the observations presented here also apply to simultaneous learning modes (*i.e.* all the steps occur at the same time).

A typical two input node is shown in figure 4.1. It receives the input signals $x_{1\rightarrow j}(t)$ and $x_{2\rightarrow j}(t)$ and generates the output signal $x_{j\rightarrow}(t)$. The node has two input weights, $w_{1j}$ and $w_{2j}$, and a weight associated with the threshold, $w_{\theta j}$.

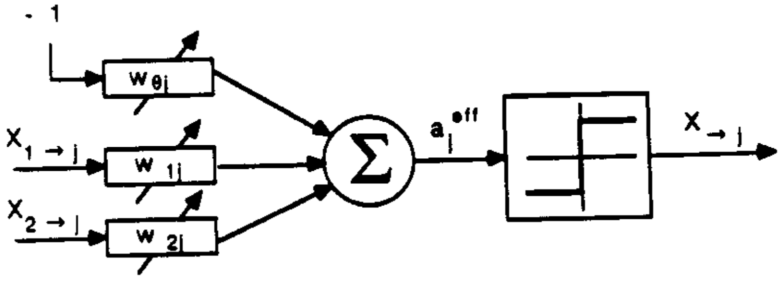


Figure 4.1: Two Input Network Node

The node's activation follows from equation 3.1:

$$a_j(t) = [w_{1j}x_{1\rightarrow j}(t) + w_{2j}x_{2\rightarrow j}(t)] \tag{4.1}$$

and the effective activation is:

$$a_j^{eff}(t) = [w_{1j}\,x_{1\rightarrow j}(t) + w_{2j}\,x_{2\rightarrow j}(t) - w_{\theta j}(t)] \tag{4.2}$$

In order for the threshold weight update equation to mimic the input weight adjustment process, it must have feedback that includes the current threshold weight value. For this reason, the activation in the threshold weight equation is replaced by the effective activation. To simplify the following analysis, it is assumed that the activation is replaced by the effective activation in all weight equations. The results will show that this assumption does not effect the average weight update vectors or the steady state weight vectors. The weight adjustment equations are therefore:

$$\Delta w_{1j}(t) = \eta(t_j^*(t) - a_j^{eff}(t))\, x_{1\to j}(t) \tag{4.3a}$$

$$\Delta w_{2j}(t) = \eta(t_j^*(t) - a_j^{eff}(t))\, x_{2\to j}(t) \tag{4.3b}$$

$$\Delta w_{\theta j}(t) = -\eta(t_j^*(t) - a_j^{eff}(t)) \tag{4.3c}$$

For now this analysis will concentrate on $w_{1j}$. Substituting equation 4.2 for the effective activation in equation 4.3a yields:

$$\Delta w_{1j}(t) = \eta(t_j^*(t) - [w_{1j}(t)x_{1\to j}(t) + w_{2j}(t)x_{2\to j}(t) - w_{\theta j}(t)])x_{1\to j}(t) \tag{4.4}$$

The change to the weight is simply the difference between the target signal and the weighted sum of the inputs, plus the threshold, all multiplied by the current input and the learning rate.

The time dependence in equation 4.4 is entirely determined by the current input vector. In other words, the change to the weights depends only on the value of the target and input signals at a given point and not the time at that point. This makes it possible to rewrite equation 4.4 in terms of the input vector:

$$\Delta w_{1j}(\mathbf{x}_{\to j}) = \eta\,(t_j^*(\mathbf{x}_{\to j}) - [\mathbf{w}_j\cdot \mathbf{x}_{\to j} - w_{\theta j}])\, x_{1\to j} \tag{4.5}$$

where $\mathbf{x}_{\to j}$ is the current input vector, namely $(x_{1\to j}, x_{2\to j})$ and $\mathbf{w}_j\cdot \mathbf{x}_{\to j}$ is the scalar product of the weight and input vectors and is equal to $(w_{1j}\, x_{1\to j} + w_{2j}\, x_{2\to j})$.

If the input pattern training set is known and the fraction that each input pattern appears is a constant throughout the training set, or the probability that an input pattern will be presented in a training set is constant, then it is possible to consider the average or expected change to the weights and threshold. The constant fraction interpretation is used when a node has already been trained or is about to be trained with a known training set, whereas the probability interpretation is used when only the probability that an input pattern will be presented at each step in a training set is known and is constant. Continuing the two input example, the average change to $w_{1j}$ can be expressed as a sum of the changes that result from each input pattern weighted by the frequency that each pattern appears. Equation 4.5 becomes:

$$\begin{aligned}\overline{\Delta w}_{1j} = \eta\,(&\alpha(-1,-1)(t_j^*(-1,-1) - a_j^{eff}(-1,-1))\,x_{1\to j}(-1,-1)\\ &+ \alpha(-1,+1)(t_j^*(-1,+1) - a_j^{eff}(-1,+1))\,x_{1\to j}(-1,+1)\\ &+ \alpha(+1,-1)(t_j^*(+1,-1) - a_j^{eff}(+1,-1))\,x_{1\to j}(+1,-1)\\ &+ \alpha(+1,+1)(t_j^*(+1,+1) - a_j^{eff}(+1,+1))\,x_{1\to j}(+1,+1))\end{aligned} \tag{4.6}$$

where

$$\alpha(-1,-1) + \alpha(-1,+1) + \alpha(+1,-1) + \alpha(+1,+1) = 1$$

and

$$x_{1\to j}(-1,x) = -1; \;\; x_{1\to j}(+1,x) = +1$$

$\alpha(p)$ is the fraction of the total number of input pattern presentations for which pattern $p$ was presented, or the probability that pattern $p$ will be presented. Note that the average change to the weight vector, expressed in equation 4.6, depends on the assumption that the activation function (which depends on the weights and threshold) does not change after each training pattern. Although this is not necessarily true, it does not affect the final results, provided that the change is small over the training set. In general, equation 4.6 can be expressed as:

$$\overline{\Delta w}_{ij} = \eta \sum_{p=1}^{2^n} \alpha(t_j^*(p) - a_j^{eff}(p))\,x_{i\to j}(p) \tag{4.7}$$

where

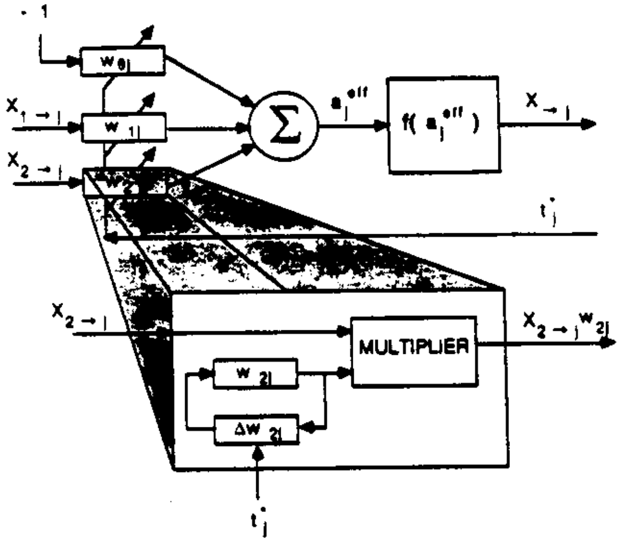


Figure 4.2: A Node With Local Feedback

provided the input pattern distribution is uniform. This means that the weight adjustment can be accomplished in parallel and without a global feedback signal (*e.g.* the node's activation). It also verifies the claim that replacing the activation with the effective activation does not change the average weight update vector because the threshold weight (added by the change to effective activation) drops out of all but the threshold weight equation. The general weight adjustment equation for a uniform input pattern distribution is:

$$\overline{\Delta w_{ij}} = \eta( \sum_{p=1}^{2^n} \frac{[x_{i\rightarrow j}(p)\, t_j^*(p)]}{2^n} - w_{ij}) \tag{4.10}$$

where $\overline{\Delta w_{ij}}$ is the average or expected change to $w_{ij}$, and $t_j^*(p)$ is the target signal for input pattern $p$. (For a two input node, the number of input patterns is equal to 4 ($2^2$).)

When the input pattern distribution is non-uniform, the weight adjustment process is not decoupled. The final weight adjustment equations for more than two inputs resemble equation 4.8 with each input weight and the threshold contributing to the average change to all other weights.

## 4.2 Steady State Values for the Weights and Threshold

In general, the steady state values for the weights and threshold are found by first setting equations 4.8a, 4.8b and 4.8c equal to zero and solving to yield:

$$w_{1j} = [\mathbf{c}_1 \cdot \mathbf{t}_j^* + \mathbf{c}_0 \cdot \mathbf{u}\, w_{2j} + \mathbf{c}_1 \cdot \mathbf{u}\, w_{\theta j}] \tag{4.11a}$$

$$w_{2j} = [\mathbf{c}_2 \cdot \mathbf{t}_j^* + \mathbf{c}_1 \cdot \mathbf{u}\, w_{1j} + \mathbf{c}_2 \cdot \mathbf{u}\, w_{\theta j}] \tag{4.11b}$$

$$w_{\theta j} = [-\frac{\mathbf{t}_j^* \cdot \mathbf{u}}{4} + \mathbf{c}_1 \cdot \mathbf{u}\, w_{1j} + \mathbf{c}_0 \cdot \mathbf{u}\, w_{2j}] \tag{4.11c}$$

Notice that if the input pattern distribution and target vector are known, the adjustment to the weights is specified by three linear algebraic equations with three unknown variables. If these equations are linearly independent, then the values of $w_{1j}$, $w_{2j}$ and $w_{\theta j}$ are uniquely determined. The final weight and threshold values for an *n* input node can thus be determined *a priori* by solving $(n + 1)$ algebraic equations. When the weight equations are not independent, however, the final values for some of the weights are under-specified and can take on a range of values. As with the weight update equations, the linear independence of these equations also depends on the values of $\mathbf{c}_0$, $\mathbf{c}_1$ and $\mathbf{c}_2$.

When the input pattern distribution is uniform, the weight update equations are necessarily linearly independent. In this case the steady state value for $w_{ij}$ is:

$$\bar{w}_{ij} = \frac{1}{2^n} \sum_{p=1}^{2^n} [x_{i \to j} \, t_j^*(p)] \qquad (4.12)$$

Note that the steady state weight values here depend only on the target and input signals.

The equations for a two input node's average weights and threshold when the input pattern distribution is uniform are:

$$\bar{w}_{1j} = \frac{1}{4} ( -t_j^*(-1,-1) - t_j^*(-1,+1) + t_j^*(+1,-1) + t_j^*(+1,+1)) \qquad (4.13a)$$

$$\bar{w}_{2j} = \frac{1}{4} ( -t_j^*(-1,-1) + t_j^*(-1,+1) - t_j^*(+1,-1) + t_j^*(+1,+1)) \qquad (4.13b)$$

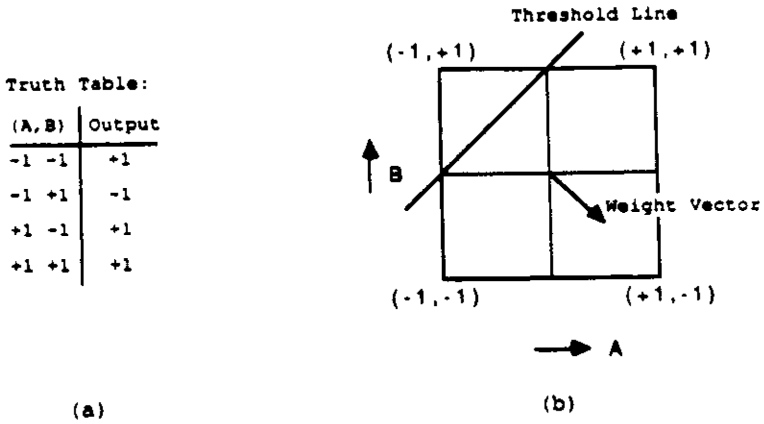
$$\bar{w}_{\theta j} = \frac{1}{4} ( -t_j^*(-1,-1) - t_j^*(-1,+1) - t_j^*(+1,-1) - t_j^*(+1,+1)) \qquad (4.13c)$$

Suppose, for example, that $t_j^*$ is described by the truth table in figure 4.3a (corresponding to the function $(A + \bar{B})$, where $A$ is associated with $x_{1 \to j}$ and $B$ is associated with $x_{2 \to j}$). The solution to equation 4.13a becomes:

$$\bar{w}_{1j} = \frac{1}{4} ( -(+1) - (-1) + (+1) + (+1)) = \frac{2}{4} = \frac{1}{2}$$

Similarly, the solutions to equations 4.13b and 4.13c become:

$$\bar{w}_{2j} = -\frac{2}{4} = -\frac{1}{2} \text{ and } w_{\theta j} = -\frac{2}{4} = -\frac{1}{2}$$



Truth Table:

| (A, B) | Output |
|---|---|
| -1 -1 | +1 |
| -1 +1 | -1 |
| +1 -1 | +1 |
| +1 +1 | +1 |

Figure 4.3: Input Space Representation of the Weights and Threshold for $(A + \bar{B})$

An input space representation of these three values is shown in figure 4.3b. In the input space representation, the $n$ inputs define an $n$ dimensional hypercube. For this example the hypercube is a square. The threshold logic unit divides the input space into two regions using an $n - 1$ dimensional hyperplane (here a line). The weights and threshold determine the orientation and location of the separation line. The orientation of the line is perpendicular to the weight vector plotted onto the input space with each weight axis lined up with the corresponding input space axis. The threshold determines how far to offset the line from the origin. Note that the separating line determined by the final weight and threshold values clearly implement the desired function.

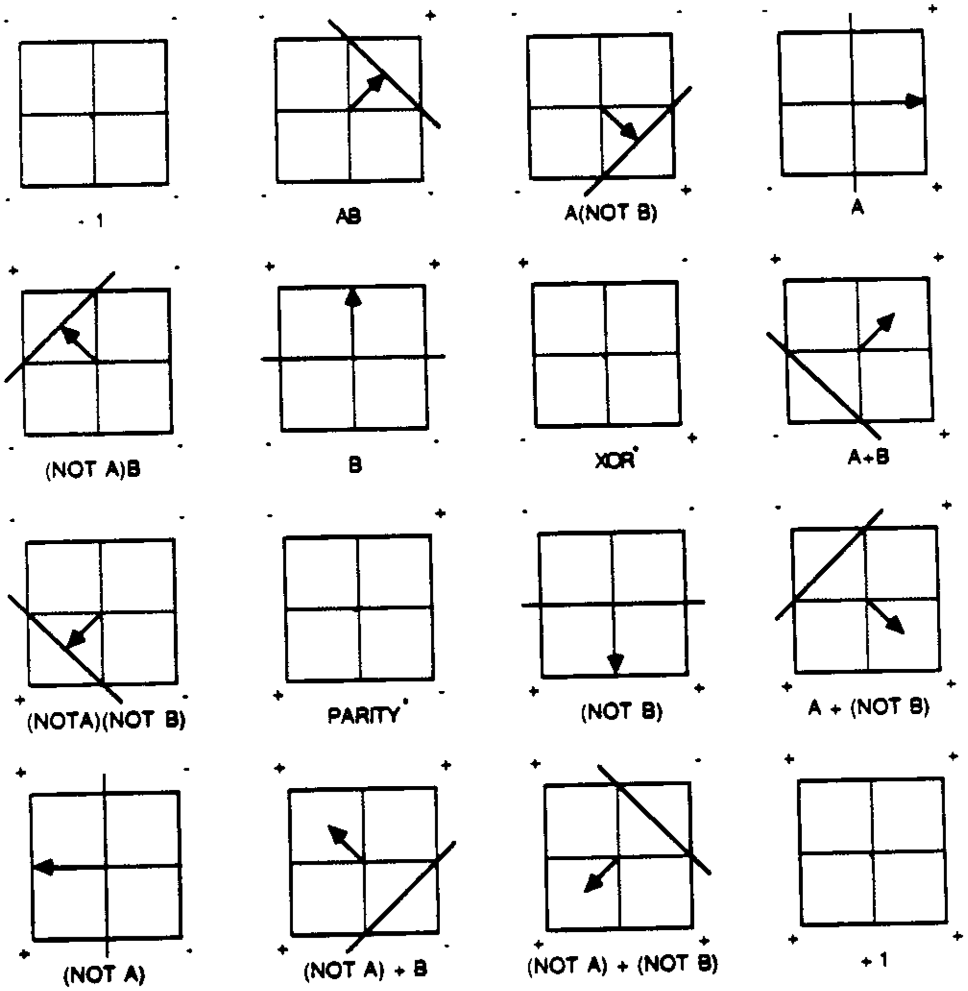
Table 4.1 contains the final weights and threshold values for a two input *AFB* node for each of the 16 possible target vectors given a uniform input pattern distribution. Notice that the *XOR* $(-1, +1, +1, -1)$ and *EQUIVALENCE* $(+1, -1, -1, +1)$ functions, which are not linearly separable (*i.e.* there is no way to divide a square with a line such that two diagonally opposite corners are on one side of the line and the other two are on the other), have solutions to the weight update equations even though these functions are not implemented by the learned weights. A graphical representation of the final weight values for each target vector is shown in figure 4.4.

| target vector | $w_{1\to j}$ | $w_{2\to j}$ | $w_{\theta j}$ | function |
|---|---|---|---|---|
| (−1, −1, −1, −1) | 0 | 0 | 1 | − 1 |
| (−1, −1, −1, +1) | + ½ | + ½ | + ½ | $A \cdot B$ |
| (−1, −1, +1, −1) | + ½ | − ½ | + ½ | $A \cdot \overline{B}$ |
| (−1, −1, +1, +1) | +1 | 0 | 0 | $A$ |
| (−1, +1, −1, −1) | − ½ | + ½ | + ½ | $\overline{A} \cdot B$ |
| (−1, +1, −1, +1) | 0 | +1 | 0 | $B$ |
| (−1, +1, +1, −1) | 0 | 0 | 0 | *A xor B* † |
| (−1, +1, +1, +1) | + ½ | + ½ | − ½ | *A or B* |
| (+1, −1, −1, −1) | − ½ | − ½ | + ½ | $\overline{A} \cdot \overline{B}$ |
| (+1, −1, −1, +1) | 0 | 0 | 0 | $\overline{A \textit{ xor } B}$ † |
| (+1, −1, +1, −1) | 0 | − 1 | 0 | $\overline{B}$ |
| (+1, −1, +1, +1) | + ½ | − ½ | − ½ | $A + \overline{B}$ |
| (+1, +1, −1, −1) | − 1 | 0 | 0 | $\overline{A}$ |
| (+1, +1, −1, +1) | − ½ | + ½ | − ½ | $\overline{A} + B$ |
| (+1, +1, +1, −1) | − ½ | − ½ | − ½ | $\overline{A} + \overline{B}$ |
| (+1, +1, +1, +1) | 0 | 0 | − 1 | + 1 |

(† *XOR* and *EQUIVALENCE* functions are not implemented by the weights and threshold)

Table 4.1 Final Weights for Two Input Generalized Perceptron

In summary, if the input pattern distribution to a network node is known and the weight adjustment equations are linearly independent, they can then be solved to determine the final values for the weights and threshold. If the input pattern distribution is uniform, then the weight equations are necessarily linearly independent and the update rules for each weight become decoupled from each other.



(* XOR and PARITY functions are not implemented by the learned weights and threshold)

Figure 4.4: Graphical Representation of Final Weights and Threshold

# Chapter 5
# Geometric Representation of Node Weight Adjustment

This chapter describes discrete learning (the simple perceptron) and linear weight adjustment (*AFB* node) processes using a geometric representation of the weights and threshold. Of particular interest is the correspondence between the threshold logic function for a given set of weights and the point that the weights map to in a *weight+threshold space*. Also of interest is the movement of the *weight+threshold* (or simply the weight) vector as a way to characterize both learning rules. Section 5.1 presents the *weight+threshold space* and its relationship to the threshold logic unit functions. This is followed by a description of the weight vector movement in response to a single training pattern and the average movement of the weight vector over a complete training set. In the last section, the steady state position of the weight vector in the *weight+threshold space* for different target functions is presented.

This chapter can be seen as a formal extension of a geometric interpretation of competitive learning that was proposed by Rumelhart and Zipser (1985) to showcase the features of their learning mechanism. Competitive learning is a method of generating target signals for hidden nodes in a network. As shown in figure 5.1, the weight vector of each node is represented by a vector whose tip lies on the surface of a hypersphere. Network learning is described as "roughly equivalent " to moving the tip of a node's weight vector along the surface of the input space sphere. The following sections will describe the formal characteristics of this representation as it applies to the weight vector of a single node.

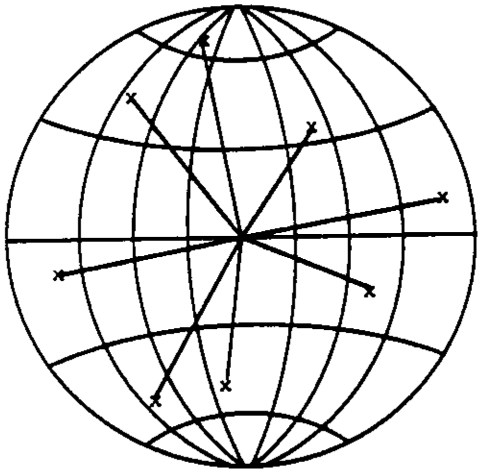


Figure 5.1: Geometric Interpretation of Competitive Learning

## 5.1 The Weight+Threshold Space

For each node, $n$ input weights plus the threshold define an $n + 1$ dimensional space referred to as the *weight+threshold space*. The current weights and threshold of a node determine a point ($w_1$, $w_2$, $w_3$, ..., $w_n$,$w_\theta$), that corresponds to the tip of a weight vector that emanates from the origin. The position of the weight vector tip determines the output function implemented by the node. This space differs from the input space representation normally used to describe node learning (as presented in section 4.2) by the addition of a dimension associated with the threshold. The function is no longer associated with a separation of the input space but with regions of the *weight+threshold space*.

Associated with a two input node is a 3-dimensional *weight+threshold cube* shown in figure 5.2. The 14 linearly separable functions of two inputs (as shown in table 4.1) carve the cube into 14 pyramid shaped regions, each corresponding to a different function. The 14 regions consist of 6 square based equilateral pyramids, one for each face of the cube, and 8 triangle based pyramids, one for each corner of the cube.

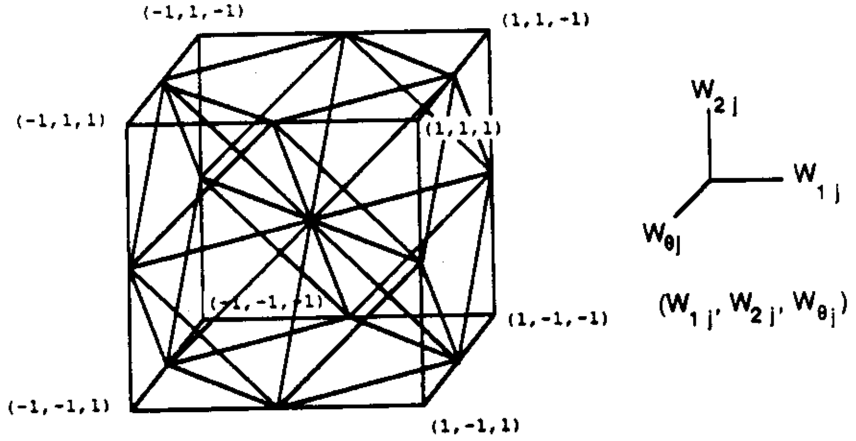
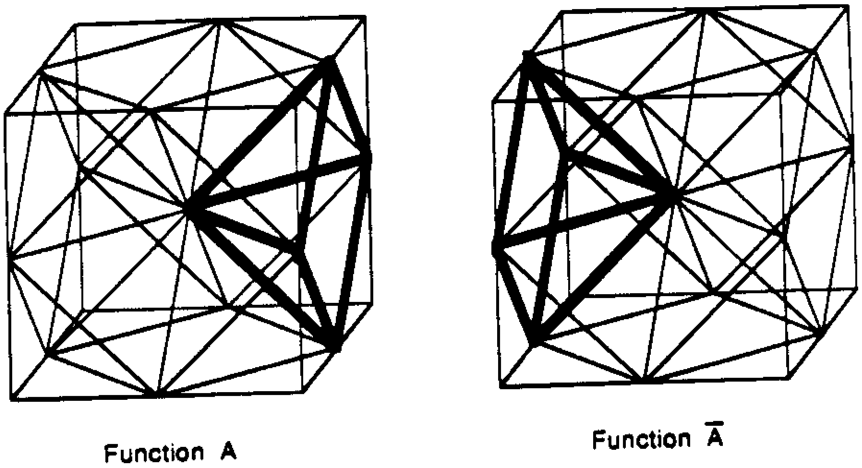


Figure 5.2: *Weight+Threshold* Cube for a Two Input Node

Figure 5.3 shows two *weight+threshold cubes* with the pyramids for the functions *A* and $\bar{A}$ highlighted. All 14 function pyramids for the two input perceptron are presented in appendix B. It is interesting to notice that two 'logically opposite' functions correspond to the physically opposite function pyramids. For example, the pyramid for the function *A* (true when A is true, false otherwise) is located with its base on the right face of the cube. The function pyramid with its base on the left face of the cube corresponds to the function $\bar{A}$ (false when A is true, true otherwise).


Function $A$

Function $\bar{A}$

Figure 5.3: Example *Weight+Threshold Space* Function Pyramids

It is also interesting to notice that all function pyramids that share a surface have truth tables that differ by only one entry (*i.e.* the truth tables differ by a hamming distance of one.) This means that as the weight vector moves from one pyramid to another, the function will change smoothly (only one truth table entry will change.) For example, the functions associated with the pyramids that share a face with the $A$ function pyramid are $(A + B)$, $(A + \bar{B})$, $(AB)$ and $(A\bar{B})$.

The origin of the *weight+threshold space* is of special interest because all function pyramids are adjacent there. The final weight vectors for the two non-linearly separable functions (*XOR* and *EQUIVALENCE*), which are not associated with a function pyramid (because they are not linearly separable), also map to the origin. Note that the actual function associated with the origin depends on how the threshold function is implemented. If the threshold comparison uses '>', then the function at the origin is '−1' or always off. On the other hand, if the threshold uses '≥' for its comparison, then the function at the origin is '+1' or always on.

In the same way, the function associated with a point on the border between two pyramids also depends on the implementation of the threshold function. If the threshold function uses '>', then the function associated with a border point is the function of the border pyramid that has the most '−1' entries in its truth table. For the case where the threshold function uses '≥', the function at a border point is the same as the border pyramid that has the most '+1' entries in its truth table. The implementation of the comparison in a discrete threshold function, therefore, determines the border cases (which includes the origin) throughout the *weight+threshold space*.

## 5.2 Weight Adjustment for a Single Training Pattern

This section will describe the effect of a single training pattern on a node's weights in terms of the movement of its weight vector. As a reminder, a training phase consists of four separate sequential steps: input pattern and target signal presentation, output generation, error calculation and weight adjustment. Also, a single training pattern consists of an input pattern and the corresponding target signal. This section will examine the effects of a single training pattern on both the simple perceptron and the *AFB* node.

### 5.2.1 Simple Perceptron

The weights of a simple perceptron are only updated when the output and target signals do not match. Recall that when the weights are adjusted, the amount of adjustment is $\pm 2\eta$ where $\eta$ is the learning rate. The simple perceptron's weights are updated using the following equation:

$$\mathbf{w}_j(t_i+1) = \mathbf{w}_j(t_i) + \Delta\mathbf{w}_j(x_{\rightarrow j}(t_i),\ t_j^*(t_i)) \tag{5.1}$$

where the weight update vector is:

$$\Delta\mathbf{w}_j = \eta(\mathbf{x}_{\rightarrow j}(t_j^* - x_{j\rightarrow}))$$

When the perceptron's output and the target signals match, $\Delta\mathbf{w}_j$ is equal to zero. When the output doesn't match the target, then the change to each coordinate of the weight vector is $\pm 2\eta$. In this case, the weight update vector for a two input simple perceptron is:

$$\Delta\mathbf{w}_j = 2\eta((x_{1\rightarrow j}\,t_j^*),\ (x_{2\rightarrow j}\,t_j^*),\ (-t_j^*)) \tag{5.2}$$

which simply reduces to:

$$\Delta\mathbf{w}_j = (\pm 2\eta,\ \pm 2\eta,\ \pm 2\eta) = 2\eta(\pm 1,\ \pm 1,\ \pm 1) \tag{5.3}$$

Notice that the direction of the weight update vector is parallel to one of the diagonals of the *weight+threshold cube* (see figure 5.4a) and is determined entirely by the input pattern and the target signal. Also notice that the weight update vector is the vector difference between two successive weight vectors as shown in figure 5.4b.

In general, the weight update vector for an $n$ input simple perceptron is an $n+1$ dimensional vector:

$$\Delta\mathbf{w}_j = 2\eta((x_{1\rightarrow j}\,t_j^*),\ldots,(x_{i\rightarrow j}\,t_j^*),\ldots,(x_{n\rightarrow j}\,t_j^*),(-t_j^*))$$

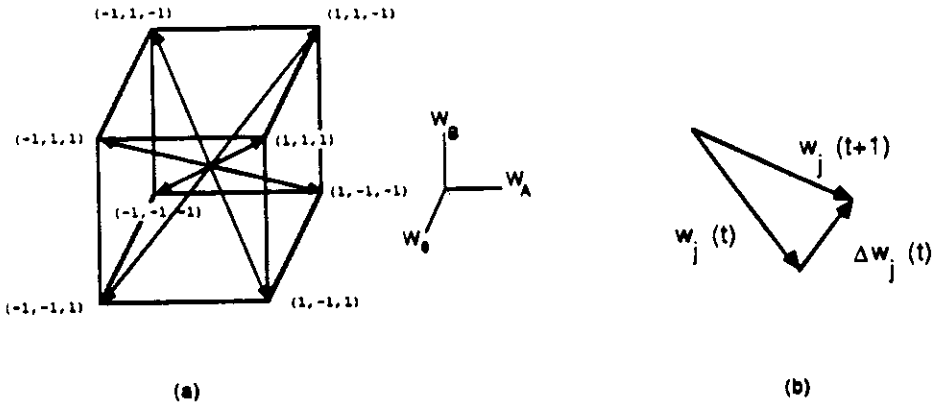


Figure 5.4: Weight Update Vector: (a) Direction Choices; (b) Vector Difference

### 5.2.2 AFB Node

The *AFB* node's linear learning rule updates the weights when the effective activation is not equal to the target signal and the amount of adjustment is proportional to the difference between the target and the effective activation. The linear learning rule can be described as:

$$\mathbf{w}_j(t_{i+1}) = \mathbf{w}_j(t_i) + \Delta\mathbf{w}_j(\mathbf{a}_j^{eff}(t_i), \mathbf{t}_j^*(t_i)) \tag{5.4}$$

and the weight update vector for a two input *AFB* node is:

$$\Delta\mathbf{w}_j = \eta(x_{1\rightarrow j}(t_j^* - a_j^{eff}), x_{2\rightarrow j}(t_j^* - a_j^{eff}), -(t_j^* - a_j^{eff})) \tag{5.5}$$

or

$$\Delta \mathbf{w}_j = \eta(1 - t_j^* \, a_j^{eff})(x_{1\to j} \, t_j^*, x_{2\to j} \, t_j^*, (-t_j^*)) \qquad (5.6)$$

Since it has been assumed that the target and input signals are either ± 1, the weight update vector can be rewritten as:

$$\Delta \mathbf{w}_j = \eta(1 \pm a_j^{eff})(\pm 1, \pm 1, \pm 1) \qquad (5.7)$$

Notice that, as with the simple perceptron learning procedure, the weight update vector is parallel to one of eight vectors that connect the origin to the corners of the *weight+threshold cube* and that the particular direction of the update vector is determined entirely by the input pattern and the target signal.

Notice that equation 5.6 and equation 5.2 are very similar. The difference between the two equations is that the magnitude of the simple perceptron's weight update vector is constant, whereas the magnitude of the *AFB* node's weight update vector depends on the current value of the node's activation. The simple perceptron and *AFB* node weight update vectors are summarized in table 5.1.

| $x_{1\to j}$ | $x_{2\to j}$ | $t_j$ | simple<br>perceptron | *AFB*<br>node |
|---|---|---|---|---|
| −1 | −1 | −1 | $2\eta\,(+1,+1,+1)$ | $\eta(1 + a_j^{eff})(+1,+1,+1)$ |
| −1 | −1 | +1 | $2\eta\,(-1,-1,-1)$ | $\eta(1 - a_j^{eff})(-1,-1,-1)$ |
| −1 | +1 | −1 | $2\eta\,(+1,-1,+1)$ | $\eta(1 + a_j^{eff})(+1,-1,+1)$ |
| −1 | +1 | +1 | $2\eta\,(-1,+1,-1)$ | $\eta(1 - a_j^{eff})(-1,+1,-1)$ |
| +1 | −1 | −1 | $2\eta\,(-1,+1,+1)$ | $\eta(1 + a_j^{eff})(-1,+1,+1)$ |
| +1 | −1 | +1 | $2\eta\,(+1,-1,-1)$ | $\eta(1 - a_j^{eff})(+1,-1,-1)$ |
| +1 | +1 | −1 | $2\eta\,(-1,-1,+1)$ | $\eta(1 + a_j^{eff})(-1,-1,+1)$ |
| +1 | +1 | +1 | $2\eta\,(+1,+1,-1)$ | $\eta(1 - a_j^{eff})(+1,+1,-1)$ |

Table 5.1 Simple Perceptron and *AFB* Node Weight Update Vectors

## 5.3 Average Weight Vector Movement

The average movement of the weight vector, or equivalently the average weight update vector, depends on the target vector, the current weight values and the fraction or frequency of input pattern presentation. This section only considers uniform input pattern distributions. Extension to non-uniform distributions follows from the non-uniform distribution analysis presented in chapter 4.

### 5.3.1 Simple Perceptron

Given the assumption that the input pattern distribution is uniform, the average movement of the weight vector is simply the movement caused by each training pattern summed up over the training set. For a two input simple perceptron, the average weight update vector is:

$$\overline{\Delta \mathbf{w}_j} = \frac{\eta}{4}\begin{bmatrix} \sum_{p=1}^{4}[x_{1\rightarrow j}(p)(t_j^{*}(p) - x_{j\rightarrow}(p))] \\ \sum_{p=1}^{4}[x_{2\rightarrow j}(p)(t_j^{*}(p) - x_{j\rightarrow}(p))] \\ -\sum_{p=1}^{4}[t_j^{*}(p) - x_{j\rightarrow}(p)] \end{bmatrix}$$

Consider the case where the target vector is (+1,+1,+1,+1). If the current output vector is (+1,+1,+1,+1) then the average weight update vector is zero. If the current output vector is (−1,−1,−1,−1) (the exact opposite of the target vector), then the average weight update vector is $2\eta(0,0,-1)$. The net change to each input weight is zero and the threshold is reduced by $2\eta$. If the current vector input has some elements that match the target vector, then only a fraction of the input patterns will contribute to the average weight update vector. Table 5.2 contains the average weight update vector for a two input simple perceptron when the output is such that all input patterns contribute to the average weight update vector.

| $\mathbf{t}_j^*$ | $\overline{\Delta \mathbf{w}}$ † |
|---|---|
| (−1,−1,−1,−1) | 2η (0, 0, 1) |
| (−1,−1,−1,+1) | 2η (½, ½, ½) |
| (−1,−1,+1,−1) | 2η (½, − ½, ½) |
| (−1,−1,+1,+1) | 2η (1, 0, 0) |
| (−1,+1,−1,−1) | 2η ( − ½, ½, ½) |
| (−1,+1,−1,+1) | 2η (0, 1, 0) |
| (−1,+1,+1,−1) | 2η (0, 0, 0) |
| (−1,+1,+1,+1) | 2η (½, ½, − ½) |
| (+1,−1,−1,−1) | 2η ( − ½, − ½, ½) |
| (+1,−1,−1,+1) | 2η (0, 0, 0) |
| (+1,−1,+1,−1) | 2η (0, − 1, 0) |
| (+1,−1,+1,+1) | 2η (½, − ½, − ½) |
| (+1,+1,−1,−1) | 2η (− 1, 0, 0) |
| (+1,+1,−1,+1) | 2η ( − ½, ½, − ½) |
| (+1,+1,+1,−1) | 2η ( − ½, − ½, − ½) |
| (+1,+1,+1,+1) | 2η (0, 0, − 1) |

† (assumes that $\mathbf{x}_{j\rightarrow} = \overline{t_j^*}$ )

Table 5.2 Average Weight Update Vectors for a Two Input Simple Perceptron

Note that while the target vector and the simple perceptron's current output function are constant, the average weight update vector is also constant. This means that while the tip of a weight vector is in the same function pyramid, it will move on the average along a straight line. When the tip crosses into a different function pyramid, the average weight vector changes and the movement of the tip is again in a straight line but with a different direction. The weight vector stops moving once its tip is just inside the function pyramid that corresponds to the target function which is when the output and target vectors are the same. In general, the average movement of the weight vector tip follows a series of line segments that end when the tip first enters the target function pyramid. An example of this is shown in figure 5.5.

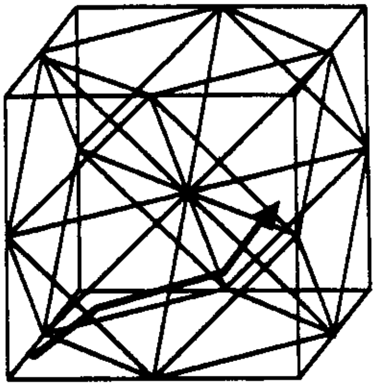


Figure 5.5: Movement of Two Input Simple Perceptron Weight Vector

In summary, the simple perceptron's average weight update vector is constant for every point within each function pyramid. The magnitude of the update vector depends on the hamming distance between the target function and the current output function where only input patterns that generate an incorrect output contribute to the average weight update vector. The tip of the weight vector follows a series of line segments that change direction at the boundaries between the function pyramids. Weight adjustment stops when the weight vector tip first enters the target function pyramid.

If the learning rate (or gain) constant is relatively small, then each adjustment to the weight vector moves its tip closer to the target function pyramid. The distance between the initial position of the weight vector tip and the target function pyramid is a finite constant and the average weight adjustment has a finite component in the direction of the function pyramid. Consequently, the simple perceptron weight adjustment process will converge on a set of weights that executes the desired function (provided it is linearly separable) in a finite amount of time. However, if the learning rate is too large, the weight vector may initially overshoot the correct target function pyramid.

### 5.3.2 AFB Node

The *AFB* node's learning rule specifies the average change to the weight vector of a two input node as:

$$\overline{\Delta \mathbf{w}}_j = \frac{\eta}{4}\begin{bmatrix} \sum_{p=1}^{4}[x_{1\to j}(p)(t_j^*(p) - a_j(p))] \\ \sum_{p=1}^{4}[x_{2\to j}(p)(t_j^*(p) - a_j(p))] \\ \sum_{p=1}^{4}[-(t_j^*(p) - a_j(p))] \end{bmatrix}$$

which is equal to:

$$\overline{\Delta \mathbf{w}}_j = \frac{\eta}{4}\begin{bmatrix} \sum_{p=1}^{4} x_{1\to j}(p)\, t_j^*(p) - 4w_{1j} \\ \sum_{p=1}^{4} x_{2\to j}(p)\, t_j^*(p) - 4w_{2j} \\ \sum_{p=1}^{4} - t_j^*(p) - 4w_{\theta j} \end{bmatrix} \tag{5.8}$$

for a uniform pattern distribution.

For the target vector (+1,+1,+1,+1), equation 5.8 reduces to: (5.9)

$$\overline{\Delta \mathbf{w}}_j = \eta( - w_{1j}, - w_{2j}, - 1 - w_{\theta j})$$

Table 5.3 contains a summary of two input *AFB* node's average weight update vector for all possible target vectors.

| $\mathbf{t}_j^*$ | $\overline{\Delta \mathbf{w}}_j$ |
|---|---|
| (−1,−1,−1,−1) | $\eta\,(- w_{1j}, - w_{2j}, 1 - w_{\theta j})$ |
| (−1,−1,−1,+1) | $\eta\,(½ - w_{1j}, ½ - w_{2j}, ½ - w_{\theta j})$ |
| (−1,−1,+1,−1) | $\eta\,(½ - w_{1j}, -½ - w_{2j}, ½ - w_{\theta j})$ |
| (−1,−1,+1,+1) | $\eta\,(1 - w_{1j}, - w_{2j}, - w_{\theta j})$ |
| (−1,+1,−1,−1) | $\eta\,(-½ - w_{1j}, ½ - w_{2j}, ½ - w_{\theta j})$ |
| (−1,+1,−1,+1) | $\eta\,(- w_{1j}, 1 - w_{2j}, - w_{\theta j})$ |
| (−1,+1,+1,−1) | $\eta\,(- w_{1j}, - w_{2j}, - w_{\theta j})$ |
| (−1,+1,+1,+1) | $\eta\,(½ - w_{1j}, ½ - w_{2j}, -½ - w_{\theta j})$ |
| (+1,−1,−1,−1) | $\eta\,(-½ - w_{1j}, -½ - w_{2j}, ½ - w_{\theta j})$ |
| (+1,−1,−1,+1) | $\eta\,(- w_{1j}, - w_{2j}, - w_{\theta j})$ |
| (+1,−1,+1,−1) | $\eta\,(- w_{1j}, -1 - w_{2j}, - w_{\theta j})$ |
| (+1,−1,+1,+1) | $\eta\,(½ - w_{1j}, -½ - w_{2j}, -½ - w_{\theta j})$ |
| (+1,+1,−1,−1) | $\eta\,(-1 - w_{1j}, - w_{2j}, - w_{\theta j})$ |
| (+1,+1,−1,+1) | $\eta\,(-½ - w_{1j}, ½ - w_{2j}, -½ - w_{\theta j})$ |
| (+1,+1,+1,−1) | $\eta\,(-½ - w_{1j}, -½ - w_{2j}, -½ - w_{\theta j})$ |
| (+1,+1,+1,+1) | $\eta\,(- w_{1j}, - w_{2j}, -1 - w_{\theta j})$ |

**Table 5.3 Average Weight Update Vectors for a Two Input *AFB* Node**

Notice that the direction of the average weight update vector for a given target vector is constant and depends only on the initial weights and the target vector. For example, if the target vector is (−1,−1,−1,−1) then the average weight update vector is

$\eta(-w_{1j}, -w_{2j}, 1 - w_{\theta j})$. This is simply $\eta$ times the difference between the steady state weight vector and the current weight vector (see figure 5.6). The average weight update movement of the *AFB* node's weight vector tip is, therefore, along a straight line which connects the tip of the initial weight vector to the tip of the steady state weight vector. An example of this is shown in figure 5.7.

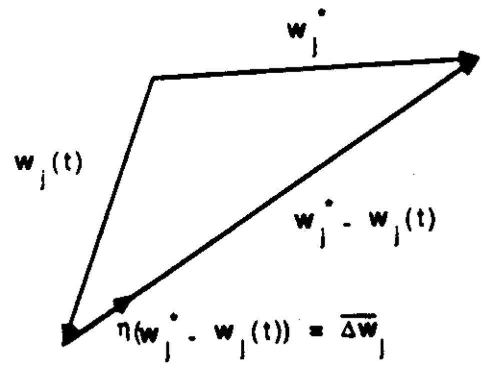


Figure 5.6: Average Weight Update Vector for an AFB Node

When the learning rate is relatively small (*e.g.* $\eta < 0.1$), the average adjustment to the weight vector moves the tip closer to the function pyramid and, unlike the simple perceptron, it continues to move when it is inside the target pyramid until the weight vector reaches its target position. Thus, a linearly separable function will be learned in a finite number of input presentations. The number of input presentations needed to learn a new output function depends on the learning rate.

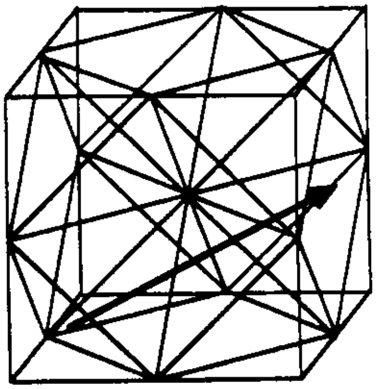


Figure 5.7: Movement of Two Input AFB Node Weight Vector

Of course, once the weight vector has reached its target position, each subsequent input presentation will move the weight vector away from the target position by a small amount that is proportional to the learning rate. Provided the input presentation remains uniform, the tip of the weight vector will stay within a neighborhood centered around the target position. The size of this neighborhood also depends on the learning rate (*e.g.* large rates may cause the neighborhood to overlap with other function pyramids).

## 5.4 Steady State Weight Vectors

This section presents a geometric interpretation of the final weight and threshold values which are dependent on the weight adjustment rule. In the simple perceptron model, the final weight vector depends on the initial weight vector and the target vector, whereas the steady state weights and threshold of a two input *AFB* node depend on the target vector and the distribution of the training set.

### 5.4.1 Simple Perceptron

When the input pattern distribution is uniform, the tip of the simple perceptron's weight vector was shown on the average to move along a series of line segments. The direction of the line segments depends on the output and target output signals. Given a target vector, the movement of the weight vector within a function pyramid can be characterized by a single update vector. The final weight vector position is determined by the initial tip position and the points where the tip moves from one function pyramid to another. Consequently, it can be any point near the surface of the target pyramid. A few final weight vectors for the target vector that correspond to the function *A* are shown in figure 5.8.

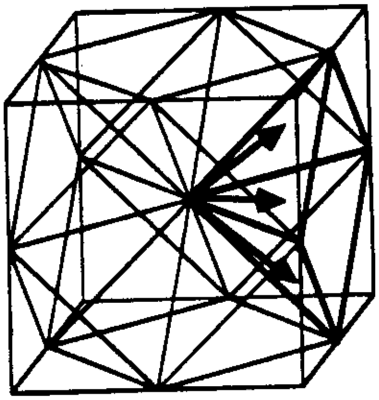


Figure 5.8: Different Simple Perceptron Final Weight Vectors

### 5.4.2 AFB Node

It was also shown that, for an *AFB* node, the average movement of the tip of the weight vector is in a straight line that connects the current tip position to the target point. A target point is a point where the average magnitude of the weight update vector is zero. When the input pattern distribution is uniform, the target points correspond to the centers of the bases of the 14 function pyramids. These uniform distribution target points are summarized in table 5.4.

| $\mathbf{t}_j^4$ | target point |
|---|---|
| (−1,−1,−1,−1) | (0, 0, 1) |
| (−1,−1,−1,+1) | (½, ½, ½) |
| (−1,−1,+1,−1) | (½, − ½, ½) |
| (−1,−1,+1,+1) | (1, 0, 0) |
| (−1,+1,−1,−1) | ( − ½, ½, ½) |
| (−1,+1,−1,+1) | (0, 1, 0) |
| (−1,+1,+1,−1) | (0, 0, 0) |
| (−1,+1,+1,+1) | (½, ½, − ½) |
| (+1,−1,−1,−1) | ( − ½, − ½, ½) |
| (+1,−1,−1,+1) | (0, 0, 0) |
| (+1,−1,+1,−1) | (0, − 1, 0) |
| (+1,−1,+1,+1) | (½, − ½, − ½) |
| (+1,+1,−1,−1) | ( − 1, 0, 0) |
| (+1,+1,−1,+1) | ( − ½, ½, − ½) |
| (+1,+1,+1,−1) | ( − ½, − ½, − ½) |
| (+1,+1,+1,+1) | (0, 0, − 1) |

Table 5.4 Target Points for a Two Input *AFB* Node

Another observation concerns the computational geometry concept of *vornoi* separation. A vornoi separation is defined for a set of *N* points as a division of a space by hyperplanes into *N* regions such that each point in the region is closer to the defining point within that region than to any other defining point. A two dimensional vornoi separation is shown in figure 5.9. Note that the lines are one dimensional hyperplanes that are equidistant from the two nearest points.

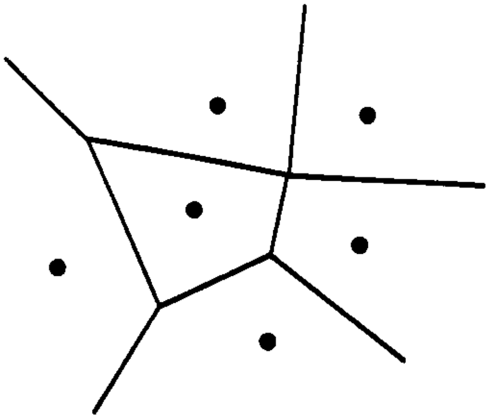


Figure 5.9: Two Dimensional *Vornoi* Diagram

An interesting observation is that the separation of the space by the threshold logic unit into function pyramids is very similiar to the *vornoi* separation of the *weight+threshold space* determined by the target points of the 14 linearly separable functions. In both cases, the areas near to the target points are associated with that point. A *vornoi-node* would have the characteristic that each function would be associated with a region whose points are all closer to the corresponding target point than to any other target point. The regions that correspond to TLU functions are different from the vornoi regions in that the simpler TLU functions (*i.e.* functions which depend on only one input such as *A*, *B*) correspond to larger portions of the space than in the vornoi separation.

## 5.5 Summary of Observations

This chapter presented a geometric representation that easily demonstrates the weight adjustment behavior of the simple perceptron and an *AFB* node. First it was shown that the movement the weight vector in both nodes, in response to a single training pattern, is in a direction that is parallel to one of the vectors that connect the origin to each corner of the *weight+threshold hypercube*. The particular direction depends on the current input and target signal. Next, the average movement of the two input simple perceptron's weight vector tip was shown to be along a series of line segments that end at a point just inside the correct function pyramid. The tip of the two input *AFB* node was shown to move on the average along the straight line connecting the current position to the point at the base of the target function pyramid. Finally, the 14 points that correspond to the steady state positions of the *AFB* node weight vectors were shown to uniquely determine the function pyramids of a TLU.

These observations can be extended to nodes with more than two inputs. For example, the response to a single training pattern will still be parallel to a vector which points to one of the hypercube corners and the tip of the simple perceptron's weight vector will still move along a series of line segments while the tip of the *AFB* node's weight vector will move along a line toward its steady state position. The correspondence between $n$-dimensional target points and $n$-dimensional function regions has not yet been explored. The problems with higher dimensional space are that it is difficult to conceptualize the correspondence between points, surfaces, and regions, and that the number of binary functions of $n$ inputs grows as $2^{2^n}$.

# Chapter 6
# A Comparison of Two Learning Rules

This chapter uses the simple perceptron and the *AFB* node to compare discrete and linear learning. The linear learning rule has a number of advantages over the discrete learning rule including an improved correspondence between the weight vector and the statistics of the training set, predictable weight movement, and immunity to noise in the weight adjustment process.

## 6.1 Improved Representation of the Training Set

One advantage of the linear learning rule is that since it uses a continuous valued feedback signal (the activation), it continues to update its weights even after the target function is learned. This provides a better representation of the training set. As shown in chapter 5, the weight vector (for a two input *AFB* node with a uniform input pattern distribution) moves to a target point that is centered within the target function pyramid; whereas, the simple perceptron's weight vector stops moving once it reaches a point which is an adequate representation of the function (*e.g.* just inside the surface of the function region). Moreover, if the input pattern distribution is not uniform, the linear weight vector will move to reflect this bias, whereas the simple perceptron's weight vector will stay constant. Thus, the final weight vector for the linear learning process provides a better representation of a training set than does the simple perceptron's.

### 6.2 Predictable Weight Movement

Another advantage to a linear learning rule is that the weights converge to a value that can be determined *a priori* from a set of equations provided that the training patterns are presented fairly. When the distribution is non-uniform, the steady state values for the weights can still be precalculated provided that the weight equations remain linearly independent. It is possible to calculate the final weights for the simple perceptron, but this requires a serial process that involves calculating the entry or exit points for each function pyramid.

When the training set is uniform, the average movement of the *AFB* node's weight vector was shown to be toward the target point. This means that the following algorithm can be used to find the weight vector that implements the target function in $2^n$ weight adjustments. First, start the weight vector at the origin. Next a complete training set is presented one at a time. After one cycle through all the training patterns, the tip of the weight vector will end up inside the correct function pyramid if the function is linearly separable. Therefore, the weights that implement the function can be found in $2^n$ time steps. This puts an upper limit of $2^n$ (where $n$ is the number of inputs) on the learning time for a linearly separable function.

This algorithm also provides, for a two input node, an $O(2^n)$ algorithm for determining whether or not a function is linearly separable. First train the weights as specified above. Then test to see if the weights implement the target function. If the weights implement the target function then the function is linearly separable, otherwise it is not linearly separable.

It has not been determined if this approach can be extended to nodes with more than two inputs. One problem with nodes that have more than two inputs is the larger number of potential target functions ($2^{2^n}$). Another problem is the fact that the average weight vector developed in chapter 4 is based on the assumption that the weights change only slightly after each training pattern and are therefore relatively constant over a complete training set. This assumption is valid when the training set is small, but may cause problems for nodes with more than 2 inputs since the size of the training set grows exponentially (*i.e.* $2^n$). More work is needed to answer some of these questions.

### 6.3 Immunity to Noise in the Linear Weight Adjustment Process

The linear weight adjustment process exhibits an inherent immunity to noise in the target signal. In contrast, the simple perceptron learning rule is very susceptible to target signal noise. This disparity results from the different ways that the two learning rules make use of the target input. The simple perceptron compares the target signal to its binary output signal and updates its weights only if the output is incorrect (*i.e.* the weight vector is on the periphery of the correct function pyramid and could be easily moved out), whereas the *AFB* node utilizes its activation to update its weights which results in improved representation of the training set (*i.e.* the weight vector tends to move toward the center of the correct function pyramid). This subtle change in the weight adjustment process has a dramatic effect. The result is that weights learned using a linear learning rule are able to implement a desired boolean function even when the target signal is corrupted with random noise.

The effect of random errors in the target signal on the output signal was measured for both the simple perceptron learning rule and the linear learning rule (used in the *AFB* node). Each trial consisted of initializing the weights followed by the successive presentation of 10,000 training patterns. After each pattern, the weights were adjusted according to the learning rule being examined. Output errors were counted each time that the actual output signal differed from the uncorrupted target signal. Trials were conducted on a two input network node for each of the 14 linearly separable functions with varying amounts of target signal error. The output errors for different amounts of target signal errors were averaged over the 14 linearly separable functions and are presented in figure 6.1. (Note: The learning rate was kept constant throughout the testing ($\eta = 0.01$).)

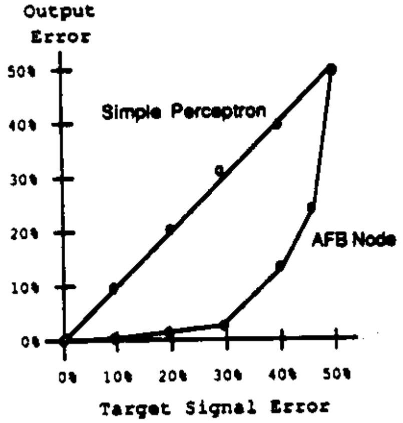


Figure 6.1: **Average Output Error Rate vs. Target Signal Error Rate.**

The results clearly show that a linear weight adjustment process is able to withstand errors in the target signal while maintaining a consistent output signal, whereas the simple perceptron learning rule produces output errors roughly as often as it receives target signal errors.

This difference in behavior can be explained by the difference in the two weight adjustment processes. The simple perceptron learning rule only updates its weights when its output is wrong (an output error). Therefore, when there is an error in the target signal, the simple perceptron's weights are updated. If the target error does not cause an output error, then the weights will not be adjusted. Thus, it is likely that an error in the target signal will lead to an output error.

The *AFB* node, on the other hand, updates its weights whenever the activation signal differs from the target signal. When a target error occurs, the input weights are updated away from their final values. The important observation here is that the next correct target signal will move the weight vector back toward its final position; therefore, the weights rebound from the target error before an output error can occur.

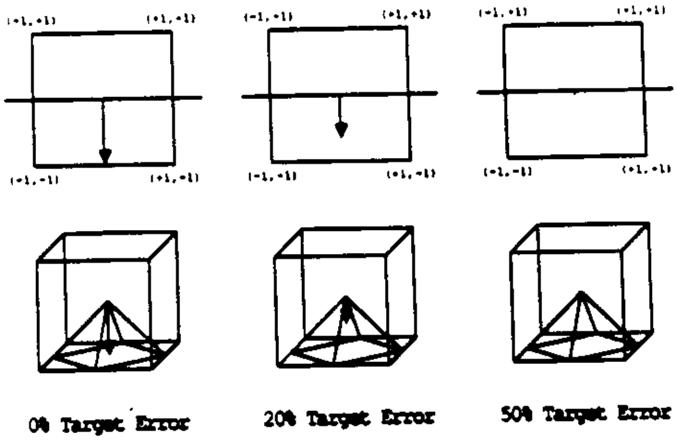
When the input pattern distribution is uniform, an expression for the final weights is

$$\bar{w}_{ij} = \frac{1}{2^n} \sum_{p=1}^{2^n} [x_{i \to j} ((1-\beta)\, t_j^*(p) + \beta\, \bar{t}_j^*(p))] \tag{6.1}$$

where $\beta$ is the percent error in the target signal and $\bar{t}_j^*(p)$ is target signal error (*i.e.* the logical opposite of the target signal). Note that $\bar{t}_j^*(p)$ is equivalent to $-t_j^*(p)$. This means that equation 6.1 can be rewritten as:

$$\bar{w}_{ij} = \frac{(1-2\beta)}{2^n} \sum_{p=1}^{2^n} [x_{i \to j}\, t_j^*(p)] \tag{6.2}$$

The effect of a constant amount of target signal noise is that the magnitude of the weight vector is reduced by a factor of twice the error rate (see figure 6.2). For example, when the error rate is 50 percent, the target signal is comparable to random noise since half of the time the target is wrong, and the magnitude of the resulting vector is zero. This make sense because half the time the tip of the weight vector will move toward the target point and half the time it will move away.



0% Target Error

20% Target Error

50% Target Error

Figure 6.2: Effect of Target Signal Error on Linear Weight Vector

# Chapter 7
# Conclusions and Future Work

This thesis has shown that a linear learning procedure is far superior to a discrete learning procedure. The comparison was accomplished using the simple perceptron and an *AFB* node which is based on an extension to the generalized delta rule. This extension provides a more flexible framework for network node design.

The linear (activation feedback) learning rule was first suggested by Widrow and Hoff (1960) as an iterative method for solving linear difference equations. Their work was concerned with moving the problem from traditional problem solving methods to a parallelizable iterative approach. It was shown in chapter 4 that although the original linear equations can be solved using traditional mathematical techniques, once they are converted to an iterative process, they can only be reduced to a set of solvable equations again when the training set guarantees that the equations remain linearly independent. It was also shown that if the training set distribution is uniform, then the linear independence condition is necessarily met.

Many advantages of a linear learning rule were presented in chapter 6. These include improved presentation of the training set statistics, more predictable weight adjustment and immunity to noise in the target signal. The *AFB* node made it possible to directly compare the performance of continuous and discrete feedback learning rules for varying levels of target signal noise. The geometric representation presented in chapter 5 makes it possible to visualize the effects of different types of feedback on the weight adjustment process.

The *AFB* node incorporates the advantages of a linear (continuous valued feedback) learning procedure into a node that communicates using binary (discrete) signals. It is therefore an interesting compromise for actual VLSI implementation because the continuous valued operations are localized. This node design may prove to be a useful approach to implementing neural networks using current VLSI techniques.

Future work on this subject will include the extension of this analysis to nodes with more than two inputs. Although the mathematical extension seems obvious, initial results have brought up some paradoxes and more work is needed. Of particular interest is extending the correspondence between weight vectors and function pyramids to higher dimensional *weight+threshold* space. Also of interest is the extension of this analysis to other non-linear threshold functions.

Future work will also include the analysis of other nodes (based on the extension to the generalized delta rule) that use a signal other than the output in the weight update process. For example, the effect of non-linear continuous threshold functions on the learning process can be examined using a similar network node.

The work in this thesis was motivated by the desire to understand in detail the dynamic behavior of neural network building blocks. It was also motivated by a desire to explore the difference between discrete and linear learning procedures. This led to an extension to the generalized delta rule and to a geometric interpretation of the weight adjustment process. It is hoped that this work will help others to understand the behavior of neural networks through a better understanding of the node learning process. Once the behavior of individual nodes is well understood, the behavior of a network of nodes will be easier to understand, and the process of building neural models will be elevated to a higher level.

# References

Amari, S. and Arbib, M. A., eds *Competition and Cooperation in Neural Nets; Lecture Notes in Biomathematics*, Springer-Berlag, Berlin (1982).

Barto, A. G., R. S. Sutton and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," IEEE Transactions on Systems, Man, and Cybernetics, **SMC-13**, No. 5, pp. 834-846, (1983).

Duda, R. O. and P. E. Hart, *Pattern Classification and Scene Analysis*, Stanford Research Institute, Menlo Park, CA (1970).

Fukushima, K., "Cognitron: A Self-Organizing Multilayer Neural Network," Biological Cybernetics, Vol. 20(3/4), pp. 121-136 (1975).

Fukushima, K., "A Hierarchical Neural Network Model for Associative Memory," Biological Cybernetics, **50**, pp.105-113, (1984).

Kan, W. and I. Aleksander, "A Probabilistic Logic Neuron Network for Associative Learning," to be published.

Hampson, S. E. and D. J. Volper, "Disjunctive Models of Boolean Category Learning," Biological Cybernetics, **56**, pp. 121-137, (1987).

Hebb, D. O., *The Organization of Behavior*, New York: Wiley, Introduction and Chapter 4, "The first stage of perception: growth of the assembly," pp. xi-xix, 60-78 in *Neurocomputing Foundations of Research*, eds J. A. Anderson and E. Rosenfeld, MIT Press, Cambridge, MA (1988).

Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Science of the USA*, Vol. 79, pp. 2554-2558 (April 1982).

Lippman, R. P., "An Introduction to Computing with Neural Nets," IEEE Acoustics, Speech, and Signal Processing Society Magazine, 4(2) pp. 4-22, (April, 1987).

Minsky, M. and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge (1969).

Nilsson, N. J., *Learning machines: Foundations of Trainable Pattern-Classifying Systems*, McGraw-Hill, New York (1965).

Pemberton, J. C. and J. J. Vidal, "When is the Generalized Delta Rule a Learning Rule? - A Physical Analogy," IEEE International Conference on Neural Networks, (1988).

Pemberton, J. C. and J. J. Vidal, "Noise Immunity of Generalized Delta Rule Learning," International Neural Network Systems Conference, (1988).

Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington, DC (1961).

Rumelhart, D. E., G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," pp. 318-362 in *Parallel Distributed Processing*, volume 1, ed. D. E. Rumelhart and J. L. McClelland, MIT Press, Cambridge, MA (1986).

Rumelhart, D. E. and D. Zipser, "Feature Discovery by Competitive Learning," pp.

151-193 in *Parallel Distributed Processing*, volume 1, ed. D. E. Rumelhart and J. L. McClelland, MIT Press, Cambridge, MA (1986).

Sutton, Richard S., and Andrew G. Barto (1981), "Toward a Modern Theory of Adaptive Networks: Expectation and Prediction," Psychological Review, No. 2, pp.135-170.

Verstraete, R. A., "Assignment of Functional Responsibilities in Perceptrons," Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA (1986).

Verstraete, R. A., "General Purpose Perceptrons: a Boolean Treatment," Master's Thesis, Computer Science Department, University of California, Los Angeles, CA (1982).

Vidal, J. J., "Silicon Brains: Whither Neuromimetic Computer Architecture," Proceedings IEEE International Conference on Computer Design, pp. 17-20, (1983).

Widrow, G. and M. Hoff (1960), "Adaptive Switching Circuits," Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4, pp. 96-104.

Winder, R. O., "Enumeration of Seven-Argument Threshold Functions," IEEE Transactions on Electronic Computers, **EC-14**, No. 3, pp. 315-325, (1965).

# APPENDIX A
# Glossary

The field of neural network research is plagued by the use of poorly defined and overused terminology (e.g. net activation to mean the activation of the network or the net amount of activation). Because of this, a few definitions are presented here in an attempt to reduce confusion. Note that the order of the terms roughly corresponds to the order in which the terms first appear in the text.

**Perceptron** refers to a class of adaptive logic units whose output(s) typically result from the application of a thresholding function to a weighted sum of the inputs, and whose weights are updated according to a prespecified learning rule.

The **simple** (or **classical**) **perceptron** is an adaptive logic based interpretation of Rosenblatt's classical perceptron model. Its weights are updated using the **perceptron learning procedure** (or **rule**.)

**The generalized delta rule** (Rumelhart *et al*, 1986) specifies a class of rules for updating the weights in a threshold logic unit. The different rules are determined by the form of the internal signal that is compared with the target signal. For example, the **linear generalized delta rule** (also called the **Widrow-Hoff rule**) compares the target signal to the (linear) weighted sum of the input signals.

The processing module of the simple perceptron and most neural network models is a **threshold logic unit** (see figure 3.1).

The processing unit of a neural network is referred to as a **functional unit, network node,** or simply **node.** The perceptron is a specific type of network node.

A **hidden node** or **internal node** is a network node that does not receive signals directly from the environment.

An **input pattern** (or **vector**) is a particular instance of the ordered set of inputs to a network or to a specific network node.

The **target signal** is the desired value for some functional unit signal (*e.g.* **target output, target activation.**) It can be provided by the environment as part of a training pattern or generated by some internal method.

A **training pattern** consists of an input pattern and the corresponding target signal.

The **training set** is the collection of training patterns used to teach a network or a specific node.

The **environment** consists of all things external to the network model, but usually refers to the teacher or other system that generates the training set.

**A node's activation** represents its current state. It is the continuous valued output of the threshold logic unit's summation module which is equal to the sum of the product of each input and its corresponding weight.

The **effective activation** is defined to be the difference between the node activation and threshold.

A node's **output** is the output of the threshold function unit. To allow for cascading the output of one perceptron or node to the input of another, the output and input signals should be in the same format.

The **activation feedback-binary (AFB) node** is a hybrid node that uses the activation signal to adjust the weights while also generating a binary output signal.

An **input pattern presentation** (also called input presentation and target pattern presentation) is the simultaneous presentation of an input pattern and a target signal.

An input pattern distribution is **fair** (or **uniform**) if each input pattern is presented equally often, or equivalently if the probability that each input pattern will be presented is equal to $1/2^n$ where $n$ is the number of inputs, namely each input pattern appears equally often. In other words, the input presentation distribution is **uniform**.

A node's **weight vector** consists of an ordered list of the current values of the input weights and threshold.

The **average weight vector** is an ordered list of the average values of the weight vector elements.

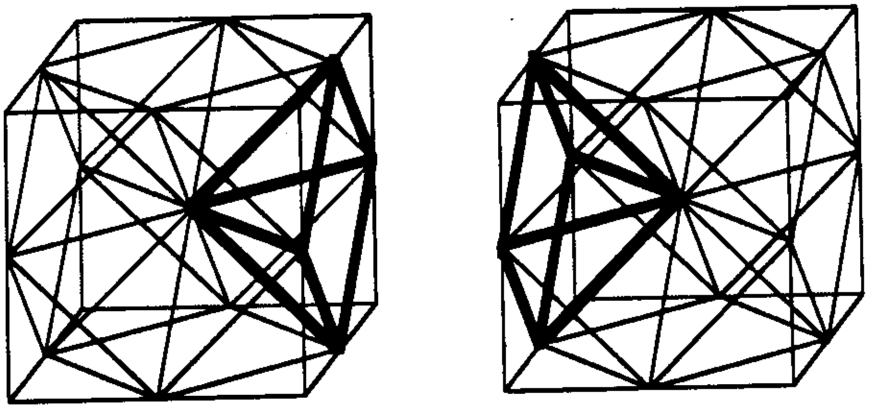
**The weight update vector** consists of an ordered list of the changes to each input weight and the threshold for a given training pattern.

The **average weight update vector** is an ordered list of the average values of the weight update vector elements.

An **error** or **target generation scheme** describes how the network model generates target or error signals for internal or hidden nodes.

# APPENDIX B
# Function Pyramids

This appendix contains a *weight+threshold space* representation for each of the 14 function pyramids that correspond to the 14 linearly separable functions of two inputs.



Function A

Function $\bar{A}$

Figure B.1: Function Pyramids for $(A)$ and $(\bar{A})$

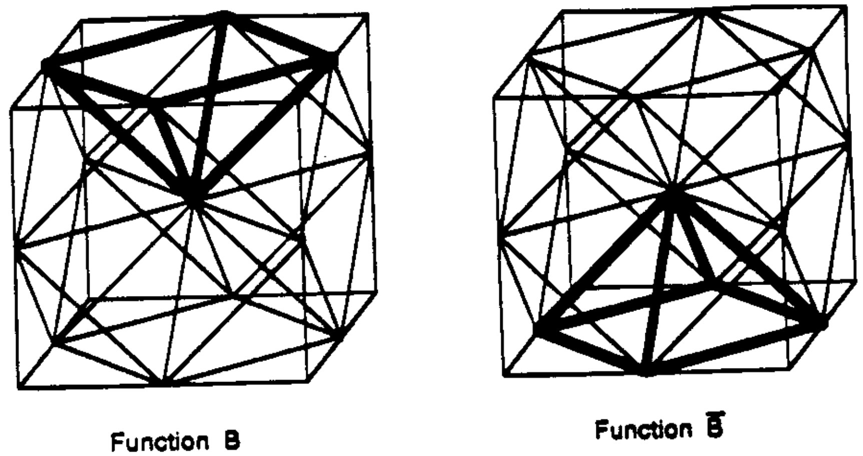


Figure B.2: Function Pyramids for ($B$) and ($\bar{B}$)

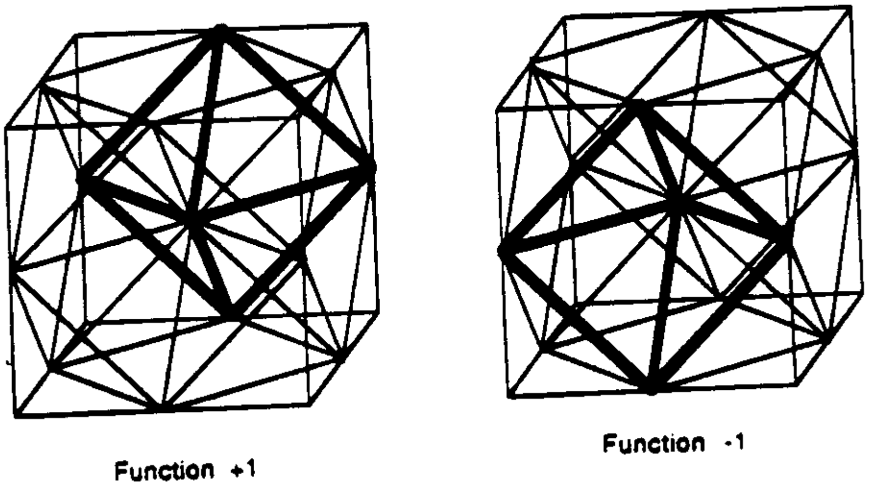


Figure B.3: Function Pyramids for (− 1) and (+1)

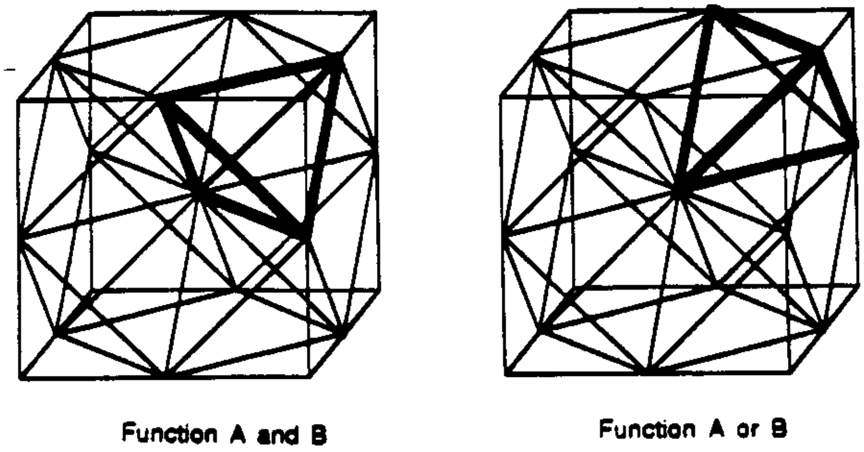


Figure B.4: Function Pyramids for (*A or B*) and (*A and B*)

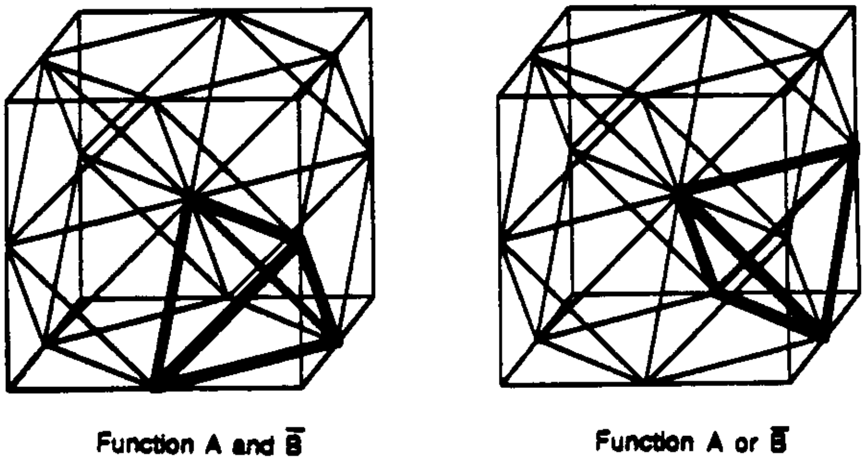
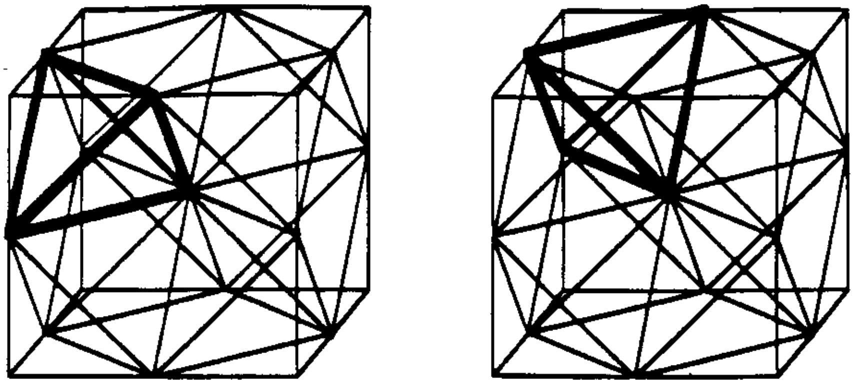


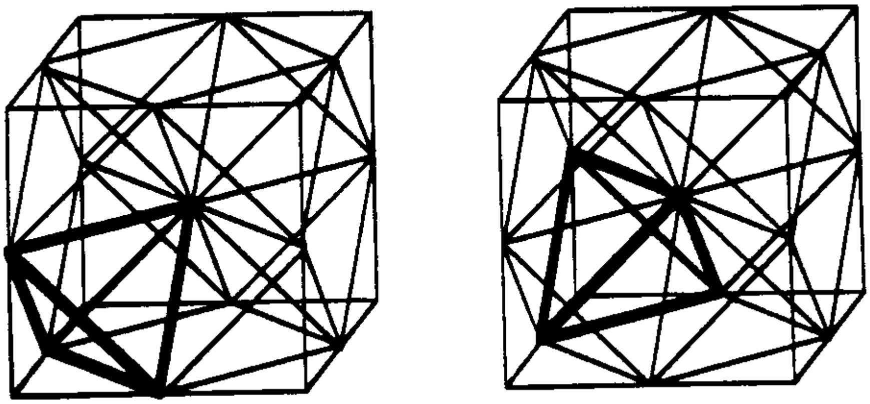
Figure B.5: Function Pyramids for (*A or* $\bar{B}$) and (*A and* $\bar{B}$)



Function $\bar{A}$ and B

Function $\bar{A}$ or B

Figure B.6: Function Pyramids for ($\bar{A}$ *or B*) and ($\bar{A}$ *and B*)



Function $\bar{A}$ and $\bar{B}$

Function $\bar{A}$ or $\bar{B}$

Figure B.7: Function Pyramids for ($\bar{A}$ *or* $\bar{B}$) and ($\bar{A}$ *and* $\bar{B}$)