ARTIFICIAL NEURAL NETWORKS

An Introduction to the Theory and Practice

by

R. C. Lacher

Professor of Computer Science

Florida State University

Version: April 24, 2009

Contact: Dr. R.C. Lacher
Department of Computer Science
Florida State University
Tallahassee, FL 32306
USA

Email: lacher@cs.fsu.edu

CONTENTS

1. Introduction.

Brief description of anatomy and function of neurons; massively parallel computations; nerve fiber = axon bundle; cerebral cortex has between 10^{10} and 10^{11} neurons and 10^4 to 10^5 connections per neuron; signal travels about 90 mph (1584 inches per second) on axon. The response time for recognition of a simple word is around 540 microseconds (.54 sec). Synapse time? Length of connection in brain? Information transfer time from cell to cell?

Given the biological hardware that has evolved (can evolve), what is the most efficient algorithm that can run on the hardware? This is the *natural algorithm* and the one we can expect to find implemented biologically.

Naturally parallel v naturally serial computations [cognize v deduce]; examples of algorithms that are naturally parallel (recognizing your grandmother) and others that are naturally serial (the connectedness problem).

cognize. (Webster) *To take cognizance of; to know, perceive, or recognize.*
(Oxford) *To know, perceive, become conscious of; to make (anything) an object of cognition.*

cognition. (Webster) *1. The process of knowing or perceiving; perception.*
2. The faculty of knowing, the act of acquiring an idea.
(Oxford) *1. The action or faculty of knowing; knowledge, consciousness; acquaintance with a subject.*
2. (phil.) The action or faculty of knowing, taken in its widest sense, including sensation, perception, conception, etc., as distinguished from feeling and volition; also, more specifically, the action of cognizing an action in perception proper.

See Table 1.1.

All these combine to prove that biological computations must be massively parallel and lead to the *neural network computational paradigm*: many computational units in a massively parallel architecture.

Watershed Events

McCulloch-Pitts (1943): Artificial neuron

Rosenblatt (1962): Perceptron learning algorithm (Hebbian learning)

Widrow-Hoff (1962): Gradient descent learning; adaptive filters

Minsky-Pappert (1969): Good theorems and bad conjectures

Werbos (1974): Backpropagation learning; economic forecasting

Hopfield (1982): Energy functions and activation dynamics; associative memory

Figure 1.1. *Schematic of generic biological neuron.*

PDP Group (1986): Connectionist models; popularization of backpropagation

Computational Intelligence (1990+): ANN integrated into a computational view of intelligence, along with symbolic AI, Bayesean nets, data mining, pattern classification, and many other previously insular avenues of interest

Organizational Dichotomies

Biological v artificial neural networks

Recurrent v acyclic networks

Supervised v unsupervised learning

The role of time

Software v hardware implementations

Scales of Space and Time

Spatial: molecules – cells – organs – organisms – societies

Temporal: See Table 1.

Modern Schools:

- Applications [engineering, economics, neurocontrol] Widrow, Meade, Werbos
- Learning and self-organization [computer science, math, physics] Kohonen, Anderson, Hopfield

Table 1.1. *Time Scale Of Human Action*‡

Scale	*Time Units*	*System*	*Organization Level*
10^7	months		
10^6	weeks		Social
10^5	days		
10^4	hours	Task	
10^3	10 min	Task	Rational
10^2	minutes	Task	
10^1	10 sec	Unit Task	
10^0	1 sec	Operations	Cognitive
10^{-1}	100 ms	Deliberate Act	
10^{-2}	10 ms	Neural Circuit	
10^{-3}	1 ms	Neuron	Biological
10^{-4}	100 μs	Organelle	

‡From Newell (1990), p. 122.

- Neuroscience modeling [biology, psychology] Grossberg, Sejnowski
- Cognitive Modelling [psychology, linguistics] PDP group, Touretzky

2. The McCulloch-Pitts Neuron.

The simplest reasonable computational model of a generic biological neuron was introduced by McCulloch and Pitts in 1943. We refer to this model as the McCulloch-Pitts neuron, or M-P neuron. The M-P neuron consists of the following components:

Terminology	**Alternate Terminology**	**Notation**
pre-synaptic inputs	inputs	$x_1, \ldots, x_m$
synaptic strengths	weights	$w_1, \ldots, w_m$
internal state	net input	y
threshold	negative bias	τ
relative input	biased net input	$\tilde{y}$
activation function	output function	φ
activation value	output	z

The components are related as follows: The internal state is the weighted sum of inputs, the biased net input is the net input minus the threshold (or plus the bias), and the output is the value obtained by applying the activation function to the biased net input. This prescription is summarized in the equations

$$y = w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{i=1}^{m} w_ix_i \tag{2.1}$$

$$\tilde{y} = y - \tau = \sum_{i=1}^{m} w_ix_i - \tau \tag{2.2}$$

$$z = \varphi(\tilde{y}) = \varphi(y - \tau) \tag{2.3}$$

The function $(x_1, \ldots, x_m) \longmapsto z$ defined by

$$z = \varphi(w_1x_1 + \ldots + w_mx_m - \tau) \tag{2.4}$$

is often called the *transfer function* or *throughput funtion* of the neuron. In general, all of these components are allowed to have any real value, although in some special cases there may be restrictions imposed. The output values will of course be restricted to the output values of the activation function. An M-P neuron is illustrated in Figure 2.1.

The components of the McCulloch-Pitts model are intended as analogous to the basic functional components of a generic biological neuron. Pre-synaptic inputs represent signals, either from outside stimuli or from other neurons. Synaptic strength

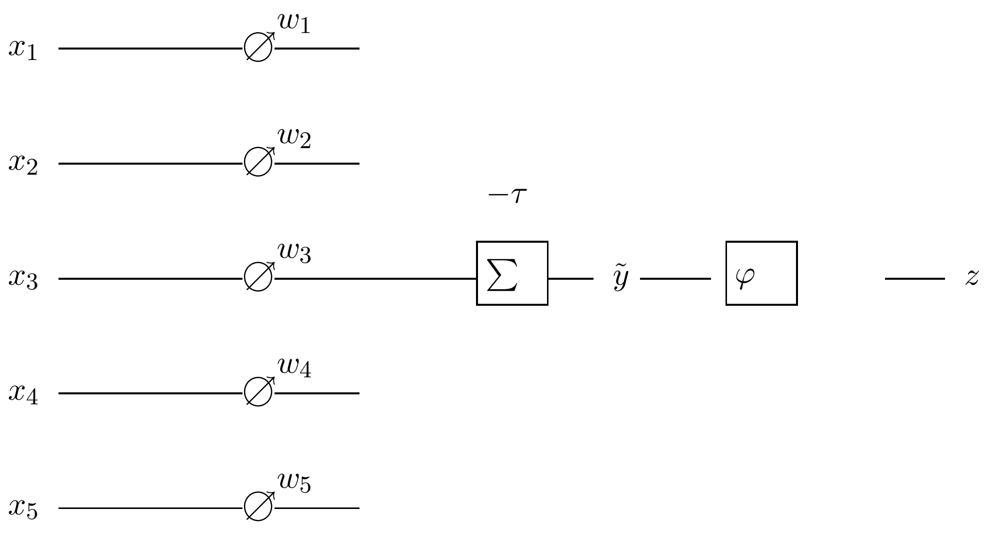

Figure 2.1. *A McCulloch-Pitts neuron with 5 inputs.*

represents the efficacy of the transmission of the input into the neuron, including membrane transport. (A negative synaptic strength represents an inhibitory connection, a positive synaptic strength represents an excitatory connection.) The internal state represents the integrated effect of all post-synaptic inputs. The output represents the firing status of the neuron.

The activation function φ for an M-P neuron can in principle be any real-valued function of a real variable. This is the only component that is not specified completely. Nevertheless, the nature of φ has considerable effect on the computational ability of networks of neurons and on their appropriateness for various applications. M-P neurons are usually classified according to the type of activation function used. We now introduce the most common types.

Threshold Units

Instead of simply passing the internal state along as the output value, as in a linear unit, a *threshold unit* outputs one of two values, interpreted usually as "firing" and "not firing", respectively. The *binary unit* uses the activation function

$$\operatorname{bin}(y) = \begin{cases} 1, & \text{if } y \geq 0; \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

Thus the output value of a binary unit is 1 if $y \geq \tau$ and 0 if $y < \tau$.

A *sign unit* uses the activation function

$$\operatorname{sgn}(y) = \begin{cases} +1, & \text{if } y > 0; \\ -1, & \text{otherwise.} \end{cases} \tag{2.6.1}$$

The output value of a sign unit is $+1$ if $y > \tau$ and -1 if $y < \tau$. An implicit requirement on a sign unit is that $y \neq \tau$, that is, that $\Sigma w_i x_i \neq \tau$, for all *allowable* inputs.

The activation functions sgn and bin are related by

$$\operatorname{sgn}(y) = 2\operatorname{bin}(y) - 1 \tag{2.6.2}$$

for $y \neq 0$.

Linear Units

A simple but very useful activation function is the *identity function* given by

$$I(y) = y. \tag{2.7}$$

An M-P neuron with the identity activation function is called an *affine unit.* If, in addition, the threshold τ is zero, we call the unit *linear* since the transfer function is a linear mapping from euclidean n-space $\mathbf{R}^n$ to the real line $\mathbf{R}$.

Sigmoidal Units

The *logistic function* is another useful and often-used activation function, given by

$$L(y) = \frac{1}{1 + e^{-\lambda y}} \tag{2.8.1}$$

where $\lambda \geq 0$ is a constant. Calculating the first derivative of $L(y)$ with respect to y and comparing the result with the product of λ, L, and $1 - L$, the following formula is verified:

$$L'(y) = \lambda L(y)[1 - L(y)]. \tag{2.8.2}$$

Using equation (2.8.2), it can be shown further that L is strictly increasing, that L takes values throughout the interval $0 < z < 1$, and that L has an inflection point at $y = \tau$ where its slope is maximized. The slope at this inflection point is $\lambda/4$, and for

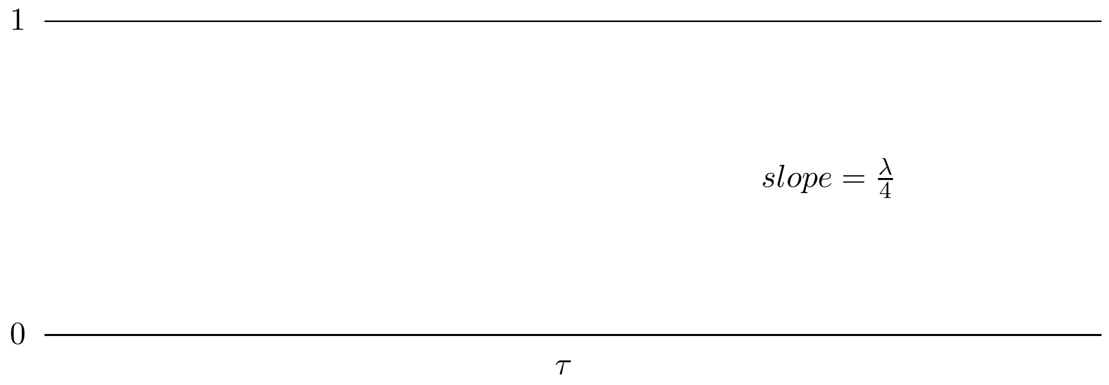

Figure 2.2. *The logistic function.*

this reason the parameter $\lambda/4$ is called the *gain* of the logistic function. These facts are summarized graphically in Figure 2.2.

It is interesting to note how the behavior of L depends on the gain parameter $\lambda/4$. As λ gets larger, the graph better approximates a step or threshold function. In the limiting "infinite gain" case, the logistic function is (up to a change of scale) the threshold sign function used as an activation function in the binary units.

Another version of the sigmoidal activation function is given by the *hyperbolic tangent function*:

$$H(y) = \tanh(\lambda y) = \frac{e^{\lambda y} - e^{-\lambda y}}{e^{\lambda y} + e^{-\lambda y}}. \tag{2.9.1}$$

The hyperbolic tangent behaves very much like the logistic function, except that the output is scaled to lie in the range $-1 < z < +1$, making H perhaps more useful in settings where output is interpreted as being on a "bad $\leftrightarrow$ good" scale instead of an "off $\leftrightarrow$ on" scale. One easily verifies the derivative formula

$$H'(y) = \lambda[1 - (H(y))^2] \tag{2.9.2}$$

from which it follows that the maximum slope of the graph of H is λ, the gain of H.

The qualitative similarity between the logistic and hyperbolic tangent functions is more than coincindence – they are related by a linear change of scale:

$$H(y) = 2L(2y) - 1 \tag{2.10}$$

which is easily verified by direct substitution. An M-P neuron whose activation function is, up to linear change of scale, the logistic function, is called a *sigmoidal unit.*

Gaussian Units

A Gaussian unit has a gaussian activation function given by

$$G(y) = \frac{e^{-(y-\mu)^2}}{2\sigma^2}. \tag{2.11}$$

This is the familiar "bell curve" of statistics, with mean μ and standard deviation σ. Normalized Gaussian units were recently introduced into neural networks as radial basis functions by Moody and Darken (1989). The effect of radial basis functions is to localize the influence of the unit within the input pattern space. Their utility has only begun to be explored.

Piecewise Linear Units

There are some activation functions in use that are neither discrete-valued nor smooth. Common examples are the linear-threshold function

$$\varphi(y) = \begin{cases} 0, & \text{if } y < \tau; \\ y, & \text{otherwise} \end{cases} \tag{2.12}$$

and the piecewise linear sigmoid function

$$\varphi(y) = \begin{cases} 0, & \text{if } y \le a; \\ 1, & \text{if } y > b; \\ \frac{y-a}{y-b}, & \text{otherwise.} \end{cases} \tag{2.13}$$

In (2.13) it is assumed that $a < b$, and the threshold is defined to be the midpoint $\tau = (a+b)/2$. These units are found in systems where smoothness is not a requirement of the learning algorithms and the piecewise linearity expedites computation or analysis.

Threshold, Bias, and Auxiliary Input

Most of the variability in the functionality of a M-P neuron is determined by the weights on inputs. Other parameters, such as threshold and gain, may also be given as inputs or as weights on clamped auxiliary inputs. The latter possibility is especially attractive for the threshold because it exposes this parameter to various learning methods that modify the weights.

The table in the beginning of this section indicates that the threshold parameter is a "negative bias". Using the notation β for bias, we have $\beta = -\tau$. The bias can be thought of as just one of the terms summed to produce the biased net input $\tilde{y}$ of the neuron:

$$\tilde{y} = \beta + \sum_{i=1}^{m} w_i x_i \tag{2.14}$$

If we set $x_0 \equiv 1$ and $w_0 = \beta = -\tau$, that is, "clamp" the aux input to the value 1 and identify the bias with the weight on that input, then we obtain

$$\begin{aligned}\tilde{y} &= \beta + \sum_{i=1}^{m} w_i x_i \\ &= w_0 + \sum_{i=1}^{m} w_i x_i \\ &= \sum_{i=0}^{m} w_i x_i.\end{aligned} \tag{2.15}$$

With this notation, we may choose to explicitly represent the threshold or bias, or we may suppress the threshold and add one extra clamped input and use unbiased neurons. The latter is often convenient in both theoretical and practical settings.

3. Network Topologies.

An artificial neural network (ANN) is a digraph whose components represent neural processes of the type discussed in Section 2. Recall that a directed graph or *digraph* consists of vertices and directed edges, and that a *network* is a digraph with a numerical value or weight associated with each directed edge. Networks are important tools for representation of flows of various types. In the case of artificial neural networks, the directed edges represent flow of signals across synapses, while vertices represent computational units:

Network	**M-P Neuron**
vertex	computational unit
edge	connection between units (synapse)
weight	strength of connection

A connection transmits the output of the pre-synaptic unit to an input of the post-synaptic unit. If a digraph represents an artificial neural network, we may refer to its vertices as *units* and its edges as *connections* to indicate that they are understood to have some functional capability.

While the nature and ability of M-P neurons varies significantly depending on the particular choice of activation function, much of the power in neural network computation is obtained by choice and variation of the digraph topology underlying the network design. (This is often called the “network topology” even though it is independent of the weights on the edges.) We now briefly consider notational and representational devices commonly used to discuss network topologies.

Just as in graph theory, it is convenient in practice to enumerate the units in an ANN. Usually, the computational ability of the ANN is completely independent of the particular enumeration chosen, so one is free to select a notational scheme that is convenient for some other purpose, be it exposition, simulation, specification, or implemenation.

There are two broad classes of ANN that are in common use today. These are mutually exclusive, but not quite exhaustive, categories based on network topology.

General Digraph Notation

A common representation scheme for digraphs simply enumerates the vertices $v_1, v_2, \ldots, v_n$ and then represents the existence of an edge from v_i to v_j by an ordered pair of subscripts. The choice of order for this pair is arbitrary but must be consistent in order to have meaning. Following IEEE suggested standards, we use the notation (j, i) to represent an edge from vertex i to vertex j. This may seem inverse from the

Figure 3.1. *Three network topologies.*

most intuitive choice, but it can be remembered by noting that the edge from v_i to v_j in an ANN represents a transmission of data and thus may ultimately lead to an assignment statement of the form $v_j := F(v_i)$.

Following this idea, suppose we are given a digraph D and an ordering $v_1, \ldots, v_n$ of its vertices. Define

$$a_{ji} = \begin{cases} 1, & \text{if there is an edge from } v_i \text{ to } v_j; \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

Then the matrix $A = (a_{ji})$ contains enough information to reconstruct the digraph. It also contains the ordering information, which may be irrelevant. The adjacency matrix is wasteful of storage in cases where D is *sparse*, i.e., when D has significantly fewer than n^2 edges (most entries of A are zero). Neverthless, this is very convenient notation for exposition, called an *adjacency matrix* representation of D. There is one for each ordering of the vertices of D.

If D is also a network, the weight matrix W is defined in a similar manner:

$$w_{ji} = \begin{cases} wt, & \text{if } wt \text{ is the weight of the edge from } v_i \text{ to } v_j; \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

If a weight of zero can be interpreted as "no connection", then the weight matrix determines the adjacency matrix in an obvious way. There are circumstances, particularly in ANN, where one may want to distinguish between "no connection" and "blocked connection" (connection with zero weight), in which case both an adjacency matrix and a weight matrix must be known.

Examples. The three digraphs depicted in Figure 3.1 have the following adjacency matrices:

$$A_1 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{3.3}$$

Layered Feed-Forward Networks

Often there are constraints placed on network topology of ANN. One of the most studied, and most useful, is the *layered feed-forward* topology, or LFF. The assumptions underlying LFF networks are as follows:

1. Units are organized into layers indexed in increasing order.
2. Connections are allowed only from one layer to the next higher layer in the index ordering.

Thus there are no intra-layer connections and no connections in the direction of decreasing layer index, but full inter-layer connectivity in the direction of increasing layer index is allowed.

Example. Suppose we have a 3-layer LFF network with units $1, 2, 3, 4, 5$ in layer 1, units $6, 7, 8$ in layer 2, and units $9, 10, 11, 12$ in layer 3. Assume that we also have full interlayer connectivity. (See Figure 3.2.) Then the adjacency matrix of this network is

Figure 3.2. *A layered feed-forward network.*

$$A = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\underline{1} & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \underline{0} & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}. \tag{3.4}$$

For example, $a_{6,1} = 1$ (underscored) indicates that output from unit 1 connects to an input of unit 6, while $a_{9,5} = 0$ (underscored) indicates no (direct) connection from unit 4 to unit 9.

Clearly there is wasted storage of 0 = 'no information' in A. Moreover, because of the LFF topology, A has a regular block structure in which all connectivity information is contained in two blocks, one for the connections from layer 1 to layer 2, and one for the connections from layer 2 to layer 3. As it becomes important, we will introduce special notation for LFF networks that takes advantage of this regular structure.

Acyclic v Recurrent Networks

A *path* in a digraph is a sequence of directed edges going from vertex to vertex in a manner so that the end of one edge is the beginning of the next in the path. A *cycle* is a path that begins and ends at the same vertex. A digraph is *acyclic* if it has no (directed) cycles. A digraph with at least one cycle is called *recurrent.* The acyclic/recurrent dichotomy is the principal watershed in most applications of digraphs, including neural networks. Recurrent topologies have "feedback loops" whereas acyclic topologies are "feedforward only".

If an enumeration $v_1, \ldots, v_n$ of the vertices of a digraph results in an adjacency matrix that is *lower diagonal*, i.e., in which all non-zero entries are below the main diagonal of the matrix, then the digraph is acyclic: connections all go from lower to higher in the vertex ordering, so no path can return to its starting vertex. An important result in the theory of algorithms says that the converse also holds: If a digraph is acyclic then its vertices can be enumerated in such a way that the adjacency matrix is lower diagonal. Intuitively, this is accomplished by placing the digraph in space and moving vertices to the right until all edges have a positive rightward component to their direction and no two have the same vertical coordinate. Then the desired ordering of the vertices is left-to-right. Such an enumeration is called a *topological sort* [Aho, et al., 1983].

For neural networks, a recurrent topology guarantees feedback among the neurons that could, and often does, reverberate forever in the network even while input is held constant. This activity is interesting and, at times, useful, as in the Hopfield associative memories and more specialized topologies considered later. On the other hand, acyclicity in a neural net frees us from worries about possibly unpredictable reverberations and permits focus on learning methods. Layered feed-forward topologies form a special case of acyclic nets that have received much attention in learning.

4. Discrete Hopfield Networks.

In 1982, John Hopfield published what is probably the single most influential paper in the field of artificial neural networks since the original work of McCulloch and Pitts. Among many other things, this work brought researchers' attention to two: activation dynamics and energy minimization techniques. While neither of these ideas was new, it was the confluence of these and other ideas that captured the attention of a wide audience. This paper more than any other event reversed the negative thinking engendered by the wrong and wrong-headed conjectures of Minsky and Papert.

A discrete Hopfield network is an ANN of threshold units that is fully connected and symmetric. That is, the adjacency matrix has every entry equal to 1, and the weight matrix $W = (w_{ji})$ satisfies

$$w_{ji} = w_{ij} \tag{4.1}$$

for each i and j. The underlying topology can be viewed as a fully connected digraph with symmetrically weighted connections, or as simply a fully connected graph with weighted 2-way (bi-directional) connections. Hopfield's original used binary units, but we find it more convenient to use sign units (see also Hertz, et al., 1991).

Thus we have n sign units indexed $1, 2, \ldots, n$. The internal state of unit j is y_j and the output of unit j is z_j. The connection strength from unit i to unit j is w_{ji}. Finally, the ith pre-synaptic input x_{ji} to unit j is the output of unit i: $x_{ji} := z_i$. This completes a discription of the *basic architecture* of a discrete Hopfield ANN. In order to augment this to a specification of a *computational network*, we need to address several issues:

How will the internal state and output value of a unit be updated?

What is the role of time in the computation?

How will the network be initialized, and how will values be retrieved from the network after computation?

Update Rule. The update of a single unit is already determined by requiring the unit be a sign unit. Applying various specifications from Section 2, we obtain the update rule

$$z_j := \mathrm{sgn}\Big(\sum_i w_{ji} z_i\Big). \tag{4.2}$$

(We assume that thresholds are zero throughout this section.)

Time. Time is discrete. The update rule is applied with centrally controlled random asynchronous updating in order to obtain activation of the entire network. Centrally controlled random asynchronous updating means that at each time step, one unit is selected at random and updated using (4.2). Locally controlled random asynchronous updating means that each unit updates itself with probability $1/n$ at each time step. These are essentially equivalent, since the probability of simultaneous updating of two units is $1/n^2$, a very small number when n is large.

Successively applying the update rule is often called *activation* of the network. We define the *activation state* of the network at a given time to be the vector $\mathbf{z} = (z_1, \ldots, z_n)$ of output values of the units. An activation state can be viewed as a pattern of ± 1 values. An activation state can be visualized as a string of lights, with $+1$ signifying 'on' and -1 signifying 'off'. The activation state (string of lights) changes over time as the system updates. At each time during asynchronous updating, exactly one unit updates, meaning exactly one component in the pattern *has the potential* of changing. The actual value may remain the same, depending on the sign of the internal state of the unit at that time. We call a pattern that is not changed by updating *stable.*

I/O. The system is initialized by "presenting" a pattern $\zeta = (\zeta_1, \ldots, \zeta_n)$ to the network, i.e., setting the activation values externally using

$$z_j := \zeta_j, j = 1, \ldots, n \tag{4.3}$$

Then the system commences updating asynchronously using the update rule.

If, at some point, the output values of the units stop changing (even though updating is still occurring), these values are retrieved and declared the output pattern $\tilde{\zeta}$.

Here is a synopsis of the process:

1. Present pattern to the network using (4.3).
2. Update the system using asynchronous updating and the update rule (4.2).
3. After the activation state of the network stops changing, retrieve this stable activation state as output.

For the remainder of this section, we say that the retrieved pattern $\tilde{\zeta}$ is *evoked* by ζ.

Associative Memory

The Hopfield network defines an association $\zeta \mapsto \tilde{\zeta}$ that can be viewed as a model of associative memory: the input pattern ζ (stimulus) evokes the output pattern $\tilde{\zeta}$ (memory). The patterns that are stable under the activation rule are stored memories. In this context, what are some properties of associative memory that would be desireable?

A stored memory μ should evoke itself. (μ is an *equilibrium.*)

If the input stimulus ζ is nearby a stored memory μ then ζ should evoke μ. (μ is an *attractor.*)

We will see that the key to these properties lies in the connections, or more specifically the connection strengths in the network.

Remembering One Pattern

Suppose we have a pattern $\mu = (\mu_1, \dots, \mu_n)$ to embed in a Hopfield network. Define the weight between units i and j to be

$$w_{ji} = \frac{1}{n}\mu_j\mu_i. \tag{4.4}$$

This defines a Hopfield network of n units.

We now want to test this network as a memory for the pattern μ. First, initialize with the pattern μ and apply the update rule (4.2) to unit j:

$$\begin{aligned} z_j &:= \operatorname{sgn}\Big(\sum_i w_{ji}\mu_i\Big) \\ &= \operatorname{sgn}\Big(\sum_i \frac{1}{n}\mu_j\mu_i^2\Big) \\ &= \operatorname{sgn}\Big(\sum_i \frac{1}{n}\mu_j\Big) \\ &= \operatorname{sgn}\Big(\frac{1}{n}\mu_j\sum_i 1\Big) \\ &= \operatorname{sgn}(\mu_j) \\ &= \mu_j. \end{aligned} \tag{4.5}$$

This shows that the value of unit j *does not change* during update, i.e., the pattern μ is an equilibrium (property 1).

Now suppose the network is initialized with another pattern ζ. Then when unit j is updated, its internal state is given by

$$\begin{aligned} y_j &:= \sum_i w_{ji}\zeta_i \\ &= \frac{1}{n}\sum_i \mu_j\mu_i\zeta_i \end{aligned}$$

$$= \frac{1}{n}\mu_j\Big[\sum_{i\in C}\mu_i^2 - \sum_{i\in I}\mu_i^2\Big]$$
$$= \frac{1}{n}\mu_j\Big[(\#\text{ correct}) - (\#\text{ incorrect})\Big] \tag{4.6}$$

where C consists of the "correct" indices i where $\zeta_i = \mu_i$ and I consists of the other "incorrect" indices. The key to understanding these steps is that all values are ±1, so either $\zeta_i = \mu_i$ or $\zeta_i = -\mu_i$. Now, consideration of (4.6) shows that if more than half of the components of ζ agree with the corresponding components of μ, then $z_j = \mu_j$ after updating unit j. After all units have been updated at least once, we have $\mathbf{z} = \mu$. Thus μ is evoked by any pattern that agrees with μ in more than half its components. In particular, μ is an attractor (property 2).

Note that the opposite of μ given by $-\mu = (-\mu_1, \ldots, -\mu_n)$ also satisfies properties 1 and 2. This "spurious" memory is retrieved from any stimulus ζ that disagrees with μ more than half the time (i.e., that agrees with $-\mu$ more than half the time).

The so-called *Hamming distance* between two n-bit patterns ξ and ζ is the number of coordinates in which the two patterns differ. We can rephrase the observations in terms of Hamming distance as follows: An input pattern ζ retrieves either μ or $-\mu$ depending on which is closer to ζ in Hamming distance. We leave it as an interesting point to contemplate what happens when these are equal.

Remembering Many Patterns

In order to extend the investigation of the previous paragraphs, suppose now that we have p patterns $\mu^1, \ldots, \mu^p$. The definition of connection strength given in (4.4) extends to

$$w_{ji} = \frac{1}{n}\sum_{q=1}^{p}\mu_j^q\mu_i^q. \tag{4.7}$$

A calculation shows that

$$\sum_i w_{ji}\mu_i^q = \mu_j^q + \frac{1}{n}\sum_i\sum_{r\neq q}\mu_j^r\mu_i^r\mu_i^q. \tag{4.8}$$

If the double sum on the right side of (4.8) is less than n in absolute value, then the entire term cannot change the sign of the first term μ_j^q, and hence $z_j := \operatorname{sgn}(\sum_i w_{ji}\mu_i^q) = \mu_j^q$. Thus, if the number p of patterns is relatively small, each memory stored by (4.7) is an equilibrium (property 1). Also, if the double sum is small, and if only a few bits of the pattern are changed, then the double sum will still be small, so the updated output returns to the value μ_j^q, so the stored patterns are attractors (property 2).

The Energy Function

Energy minimization principles are common in physics. The introduction of them into the study of neural networks was one of the most valuable contributions of Hopfield's paper. Given a Hopfield network of n units, define the *energy* of the network at state $\mathbf{z}$ to be the value of

$$H(\mathbf{z}) = -\frac{1}{2}\sum_{ji} w_{ji} z_j z_i. \tag{4.9}$$

By the symmetry property (4.1), we have

$$\begin{aligned} H(\mathbf{z}) &= -\sum_{j} w_{jj} z_j^2 - \frac{1}{2}\sum_{j\neq i} w_{ji} z_j z_i \\ &= -D - \sum_{j<i} w_{ji} z_j z_i \end{aligned} \tag{4.10}$$

where $D = \sum_j w_{jj}$. Consider the effect on energy of updating unit k:

$$z'_k := \operatorname{sgn}\Big(\sum_i w_{ki} z_i\Big). \tag{4.11}$$

If the updated value z'_k is equal to the old value z_k, clearly the energy computation (4.9) does not change. If however $z'_k \neq z_k$ then $z'_k = -z_k$, and we calculate the energy change:

$$\begin{aligned} H' - H &= -\sum_{j<i} w_{ji} z'_j z'_i + \sum_{j<i} wjiz_j z_i \\ &\qquad \text{(all terms cancel except } j = k, i = k\text{)} \\ &= -\sum_{j\neq k} w_{jk} z'_j z'_k + \sum_{j\neq k} w_{jk} z_j z_k \\ &\qquad (z'_j = z_j \text{ for } j \neq k) \\ &= -\sum_{j\neq k} w_{jk} z_j z'_k + \sum_{j\neq k} w_{jk} z_j z_k \\ &= \sum_{j\neq k} w_{jk} z_j z_k + \sum_{j\neq k} w_{jk} z_j z_k \\ &= 2z_k \sum_{j\neq k} w_{jk} z_j \\ &= 2z_k \sum_{j} w_{jk} z_j - 2w_{kk} z_k^2 \\ &= 2z_k \sum_{j} w_{jk} z_j - 2w_{kk}. \end{aligned} \tag{4.12}$$

In the last expression, the first term is negative because $z'_k := \operatorname{sgn}(\sum_j w_{jk} z_j) = -z_k$, and the second term is negative because $w_{kk} = \sum_q (\mu_k^q)^2 = p/n$. This proves the following

Fact. *The energy H decreases whenever updating changes the activation state of the network.*

This energy principle allows us to view activation as a dynamical system that uses a finite amount of energy each time it changes state. Since there are only finitely many states, eventually the state reaches a level of energy that cannot be decreased. Thus any initialization (input stimulus) eventually reaches a memory state (a local minimum of energy). Moreover, (4.8) shows that memories stored using (4.7) are such local energy minima.

Memory Capacity

The issue of exactly how many patterns can be stored stably (i.e., as local minima of energy) in a Hopfield memory was glossed over in the discussion following (4.8). Let p_{max} denote the "maximum" number of patterns the network can store stably. What is p_{max}?

Answer 1: $p_{max} \sim n$ if we accept a small percent error in each retrieval, that is, if we accept that all but a few bits in the retrieved memory are correct.

Answer 2: $p_{max} \sim \frac{n}{\log n}$ if we insist that most of the stored patterns are recalled exactly.

See Hertz, et al. (1991) for a more complete exposition.

Spurious Memory States

When memories are stored in a Hopfield network using (4.7), there are also spurious, or unwanted, memories created. The so-called odd-mix states are examples. Given one memory state μ, its opposit $-\mu$ is a spurious memory state, as we observed earlier. This is a 1-mix state.

For three memories μ^1, μ^2, μ^3, the 3-mix states are the states $\mu^{\pm\pm\pm}$ defined by the equations

$$\mu_j^{\pm\pm\pm} = \operatorname{sgn}(\pm\mu_j^1 \pm \mu_j^2 \pm \mu_j^3). \tag{4.13}$$

The signs are held consistent for $j = 1, \ldots, n$, so there are eight 3-mix states specified by (4.13). Note, for example, that μ^{+++} has Hamming distance $n/4$ from μ^1 because $\mu_j^{+++} \neq \mu_j^1$ one out of four times. For odd numbers greater than one, the odd-mix states are memory states (local energy minima) and have higher energy levels than the pure states μ^1, μ^2, μ^3.

Figure 4.1. *Activation state space.*

Figure 4.2. *Energy versus activation state.*

There are two graphical depictions of this situation. In Figure 4.1, the plane is representative of the activation state space of a Hopfield network. Three memory states are indicated, along with one mix state. The regions delineated by the curves represent the basins of attraction of the four local energy minima. In Figure 4.2, the horizontal scale represents activation state space and the vertical scale represents energy level.

5. Functional Capacity of Threshold Units.

Having taken a glimpse into the world of neural networks and collective computational behavior, we return to study individual neurons of the type used in Hopfield nets.

Recall from Section 2 that a *sign unit* consists of the following components and relations:

Name	**Notation**	**Allowed Values**
inputs	$x_1, \ldots, x_m$	± 1
weights	$w_1, \ldots, w_m$	real
net input	y	real
threshold	τ	real
biased net input	$\tilde{y}$	real
output	z	± 1

$$y = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \sum_{i=1}^{m} w_i x_i \tag{5.1}$$

$$\tilde{y} = y - \tau = \sum_{i=1}^{m} w_i x_i - \tau \tag{5.2}$$

$$z = \operatorname{sgn}(\tilde{y}) = \operatorname{sgn}(y - \tau) \tag{5.3}$$

where $\operatorname{sgn}(\tilde{y})$ denotes the sign function whose value is $+1$ if $\tilde{y} > 0$ and -1 if $\tilde{y} < 0$. Alternate terminology for a sign unit is *binary neuron.* Note that

$$z = \operatorname{sgn}(\tilde{y}) = \frac{\tilde{y}}{|\tilde{y}|} \tag{5.4}$$

where $|\tilde{y}|$ denotes the absolute value of $\tilde{y}$. This last equation emphasizes that z is *not defined* when $\tilde{y} = 0$. Therefore an implicit requirement on a sign unit is that $y \neq \tau$, that is, that $\Sigma w_i x_i \neq \tau$, for all possible binary inputs.‡ A sign unit is illustrated in Figure 5.1.

‡ The output equation (5.3) can be enhanced by defining $z = +1$ whenever $y = \tau$, avoiding this restriction. For purposes of analysis, retention of the restriction emphasizes the critical nature of the "at threshold" input. The inherent robustness of sign units makes this point of little practical significance.

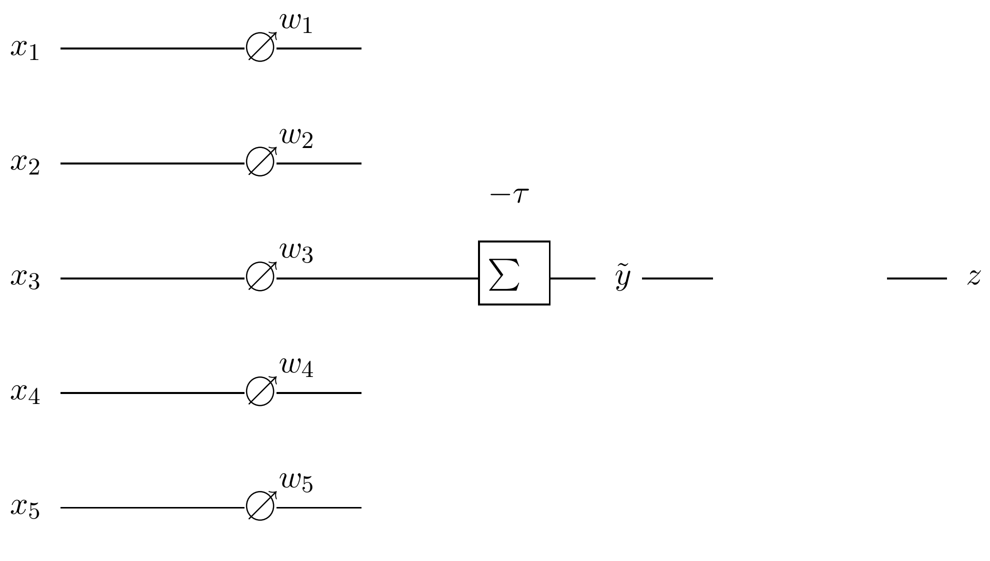

Figure 5.1. *A sign unit with 5 inputs.*

There are two views of a binary neuron, external and internal. Imagine that a shell or membrane separates the external world from the internal structure of the unit. The external world is discrete, and here a unit appears to be a binary-valued function of several binary variables (inputs) whose nature is "tunable" by adjusting several parameters (weights). (This function is called the *transfer function* of the neuron.) The unit is a "black box", however, and from the external discrete world the exact nature of the unit is not discernable. To understand how the sign unit transforms input to output, and to know exactly which binary function a given setting of the weights defines, we must pierce its shell or membrane, dissecting the unit so to speak. Inside the membrane we find an analog world and a precisely described mechanism with which we can completely describe the external function of the unit.

Linear Separation

A geometric interpretation of how output is computed may be obtained by getting inside the sign unit, into the analog (real) world. The value y is a *real* number that can be computed for any *real inputs* $x_1, \ldots, x_m$. The equation $y = \tau$ expands to a linear equation whose solutions are the "illegal" inputs:

$$w_1x_1 + \ldots + w_mx_m = \tau. \tag{5.5}$$

Equation (5.5) defines a hyperplane in real $(x_1, \ldots, x_m)$-space $\mathbf{R}^m$ called the *threshold hyperplane*. This hyperplane separates $\mathbf{R}^n$ into two components, one in which $\tilde{y}$ has positive values and the other in which $\tilde{y}$ has negative values. The threshold hyperplane, together with knowledge of which component is the positive side, com-

a. *Linear separation.*

b. *Finding a separator.*

Figure 5.2.

pletely determine the binary function defined by the unit: $z = +1$ for inputs on the positive side and $z = -1$ for inputs on the negative side.

The Case $m = 2$. Consider the special case of a sign unit with two inputs. Then $\mathbf{R}^{n=2}$ is the euclidean (x_1, x_2)-plane. Equation (5.5), which specializes to

$$w_1 x_1 + w_2 x_2 = \tau, \tag{5.5.1}$$

defines the threshold line (hyperplane) that separates $\mathbf{R}^2$ into a '+' side and a '−' side. The four points in the plane whose coordinates are ± 1 represent the possible binary inputs. As long as the threshold line given by (5.5.1) does not intersect these four points, z is defined and is the sign of $\tilde{y}$, that is, $z = +1$ for inputs on the '+' side and $z = -1$ for inputs on the '−' side. See Figure 5.2.

A line such as given by (5.5.1) can be plotted in the (x_1, x_2)-plane by calculating the intersections with the coordinate axes. (Recall these intersection points are called *intercepts*. The x_2-intercept, for example, is found by setting $x_1 = 0$ and solving (5.5.1) for x_2.) If the line is not horizontal (i.e., if $w_1 \neq 0$) then the x_1-intercept is τ / w_1, and if the line is not vertical (i.e., if $w_2 \neq 0$) then the x_2-intercept is τ / w_2. If the line is not vertical, (5.5.1) can be rearranged into the *slope-intercept* form

$$x_2 = m x_1 + b \tag{5.5.2}$$

where $m = \text{slope} = -w_1 / w_2$ and $b = x_2\text{-intercept} = \tau / w_2$.

Example 1. Suppose $n = 2$ and the parameters have values $w_1 = +1, w_2 = -2, \tau = +2$. Then the threshold line has slope $1/2$ and x_2-intercept -1. The line is illustrated in Figure 5.2a. From inspection of that figure, we see that the binary inputs $(+1,+1), (-1,+1)$, and $(-1,-1)$ all lie on one side of the threshold line and $(+1,-1)$ lies on the other. To know what binary function this particular unit defines, we need one more item of information: which side is the '+' side of the threshold line. This is easy to discover: just test the function $\tilde{y} = w_1x_1 + w_2x_2 - \tau$ on some convenient set of input values such as $(x_1, x_2) = (+1,+1)$:

$$\begin{aligned} \tilde{y}|_{(+1,+1)} &= w_1x_1 + w_2x_2 - \tau \\ &= w_1 + w_2 - \tau \\ &= (+1) + (-2) - (+2) \\ &= -3 \qquad (5.5.3). \end{aligned}$$

Thus $\tilde{y}$ is negative on $(+1,+1)$ and, hence, on the entire half-plane containing that point. And, $\tilde{y}$ must be positive on the other half-plane. Inspection of Figure 5.2a now shows that $z = \text{sgn}(\tilde{y})$ is the binary function that is $+1$ on $(+1,-1)$ and -1 on $(+1,+1), (-1,+1)$ and $(-1,-1)$. ▮

The process illustrated in Example 1 can be reversed: given a binary function, find what weight values realize that function as a sign unit. If we take $+1$ to mean 'true' and -1 to mean 'false', the question can be rephrased in terms of logical functions.

Example 2. Suppose we want to realize the logical NAND operation as a sign unit. NAND is specified by the following table of values (truth table):

NAND		
x_1	x_2	$x_1\|x_2$
$+1$	$+1$	-1
$+1$	-1	$+1$
-1	$+1$	$+1$
-1	-1	$+1$

We begin by finding the binary input pairs in the (x_1, x_2)-plane and labelling them with the desired binary output values, as illustrated in Figure 5.2b. Then find a (convenient) line that separates the '+' value from all of the '$-$' values. One such line is illustrated in the figure. That line passes through the points $(0,+1)$ and $(+1,0)$, so it has slope -1 and x_2-intercept $+1$. Plugging this information into equation (5.5.2) and rearranging terms results in the equation

$$x_1 + x_2 = 1. \qquad (5.5.4)$$

Comparing coefficients between (5.5.4) and (5.5.1) results in a *trial* set of values for the parameters: $w_1 = 1, w_2 = 1, \tau = 1$. These in turn determine a *trial* definition of the function $\tilde{y}$:

$$\tilde{y} = x_1 + x_2 - 1. \tag{5.5.5}$$

Either these make a correct set of parameters or their negatives do; we find out which by testing the value of $\tilde{y}$ on a convenient input. Testing $\tilde{y}$ on the input $(+1, +1)$ yields

$$\begin{aligned} \tilde{y}|_{(+1,+1)} &= w_1 + w_2 - \tau \\ &= (1) + (1) - (1) \\ &= +1 \end{aligned} \tag{5.5.6}$$

which is not the correct sign. Therefore we reverse the signs of each of the trial parameters, obtaining

$$\tilde{y} = -x_1 - x_2 + 1 \tag{5.5.7}$$

which correctly realizes the NAND function. A similar analysis shows that the logical AND and OR functions are also realized by sign units. ▮

The General Case. The analysis of sign units using the threshold hyperplane works in the general case of m input variables in much the same way. The threshold hyperplane separates $\mathbf{R}^m$ into two half-spaces. The external binary inputs must not lie on the plane, so they each lie in one of the two half-spaces. The biased net input function $\tilde{y}$ is positive on one of the half-spaces and negative on the other, and the '+' side can be determined by evaluating $\tilde{y}$ on the input $(+1, \ldots, +1)$, i.e., by adding up all of the weights and subtracting the threshold:

$$\tilde{y}|_{(+1,\ldots,+1)} = \sum_{i=1}^{m} w_i - \tau. \tag{5.6}$$

The sign of the number calculated by (5.6) indicates which half-space contains the input point $(+1, \ldots, +1)$. The binary function z has the value $+1$ at each input that is in the '+' half-space and -1 at each input that is in the '$-$' half-space.

Conversely, consider an arbitrary binary-valued function β of m binary variables $x_1, \ldots, x_m$. We say that β is *linearly separable* if there exists a hyperplane H in $\mathbf{R}^m$ that separates the binary inputs $(x_1, \ldots, x_m)$ according to their value assigned by β: $\beta(x_1, \ldots, x_m) = +1$ for all inputs on one side of H and $\beta(x_1, \ldots, x_m) = -1$ for all inputs on the other side of H. An argument similar to the one outlined in Example 2 shows that if β is linearly separable then it is realizable by a sign unit. We summarize these conclusions as follows:

a. No linear separation. *b. Separation by two lines.*

Figure 5.3.

Theorem 1. *The binary function β is realizable by a binary neuron if and only if β is linearly separable.*

The criterion of linear separability can be used to find non-realizable functions.

The XOR Problem

Consider the problem of realizing the logical exclusive-or operation with a sign unit. A truth table for exclusive-or, also called XOR, is:

XOR		
x_1	x_2	$x_1 \oplus x_2$
$+1$	$+1$	-1
$+1$	-1	$+1$
-1	$+1$	$+1$
-1	-1	-1

After graphing the inputs in the (x_1, x_2)-plane and labelling the inputs with the desired output sign, it becomes apparent that the '+' inputs cannot be separated from the '−' inputs with a (straight) line. (See Figure 5.3a.) Since a sign unit that realized the function would have a threshold line that separates the inputs, as explained in the examples, there can be no such unit. That is, *the* XOR *function*

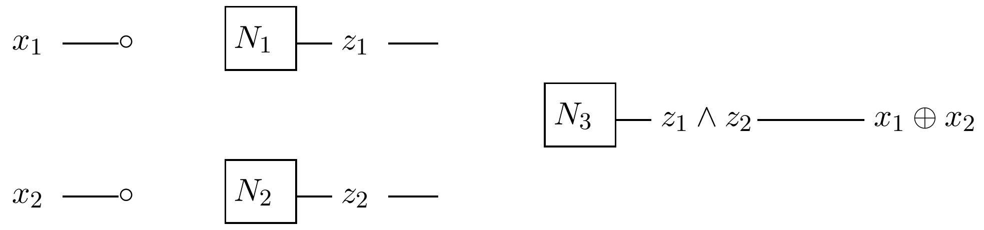

Figure 5.4. *A network of 3 sign units realizing* XOR.

cannot be realized by a sign unit. This result is a precursor to a more complex result of Minsky and Papert, discussed in a later section. But first, is there anything we can do to remedy the problem?

Yes. We can separate the inputs appropriately with *two* lines, as illustrated by L_1 and L_2 in Figure 5.3b. These lines, annotated with '+ side' indicators as in the figure, determine logical functions z_1 and z_2, respectively, that are each realizable by sign units. Values for these functions can be read directly from the geometric information displayed in the figure. A table of these values follows.

x_1	x_2	z_1	z_2
$+1$	$+1$	-1	$+1$
$+1$	-1	$+1$	$+1$
-1	$+1$	$+1$	$+1$
-1	-1	$+1$	-1

One interpretation of this table is that the two functions together can "recognize" the XOR pattern (they are both $+1$ when XOR is $+1$). It is also apparent from the table that the conjunction of the two functions is equal to XOR. That is, the function

$$z = z_1 \wedge z_2 \tag{5.7}$$

is the XOR function. At this point we have shown that the binary functions z_1 and z_2 are each realized by a sign unit. It is easy to check that the logical AND function is linearly separable and is therefore also represented by a sign unit, completing an argument for the following

Theorem 2. *The function* XOR *can be realized by a network of three binary neurons.*

The network is illustrated in Figure 5.4. We conclude with some comments.

Robustness. A sign unit is quite robust in the sense that small changes in its weights and/or threshold do not change the binary function it realizes. There are two sources for this freedom of choice of parameters, algebra and geometry. Algebraic freedom is created because, after the threshold hyperplane and '+ side' have been specified, the unit parameters are still underconstrained. In equation (5.5.7) above, for instance, we could multiply all of the parameters (weights and threshold) by any positive constant c and still describe exactly the same line. The new biased net input function would be the product of c and $\tilde{y}$, where $\tilde{y}$ is the original one. Since the product $c\tilde{y}$ has the same sign as $\tilde{y}$, the resulting binary transfer function $z = sgn(c\tilde{y})$ is unchanged.

There is also geometric freedom in the choice of the line itself. In Figure 5.2b, for example, we could shift the chosen line slightly, obtaining a different line and a different biased net input, yet, if the binary inputs are still separated in the same way, we would end up with the same binary function z. All this freedom means that sufficiently small changes, "tweeks", and errors will not affect the binary function realized by a sign unit.

Non-Linearity. History would ask that the descriptor "linear" be added to the terminology for binary neural element, but we choose to differ with history: while linear structures are used in the elegant analysis, binary neural elements are *not linear* in the universally accepted meaning of that term: they are not linear functions over either the real field or the 2-element field. In fact binary neural elements can be thought of as the "infinite gain" case of a non-linear analog neuron. Moreover, it is precisely their non-linearity that makes binary neurons useful in multi-layer networks. This point is revisited in a later section, after we have made a study of feed-forward networks of non-linear analog neurons.

The Sign Unit as a Model Neuron. Few would argue that a binary neuron is a realistic model of a biological neuron. Real neurons are incredibly complex, both in structure and behavior; they are objects of intense investigation in the neuroscience community.

It is good to keep in mind where we are heading, however. Our artificial neurons are not intended to be particularly useful devices by themselves, but rather are to be used as the nodes in a network. These neurons will be "atomic" level components in a model that derives its complexity from interconnection of many such elements. The network itself, rather than the individual nodes, is anticipated to display complex behavior. The question is one of scale. An analogy can be made with the use of models of the earth. Studying a single neuron through models is analogous to studying the earth itself, as in geology, oceanography, and meteorology, where extremely complex models of earth must be adopted. Modeling a neural network, on the other hand, is analogous to studying a celestial system, as in astrophysics, where modeling a planet as a perfect sphere is an acceptable simplification.

6. The Perceptron Learning Rule.

The concept of collective behavior is a primary organizing force in the subject of artificial neural networks. Collective behavior may be manifested in activation properties, as introduced with the Hopfield network in Section 4. It is also characteristic of learning, which we introduce here.

A *perceptron* is a layered feed-forward network of threshold units. The terminology was introduced by Rosenblatt (1962) along with the *perceptron learning rule.* This rule is a method of incrementally adjusting the weights of a simple (1-layer) perceptron so as to improve its ability to realize a given binary mapping.

A simple perceptron is a parallel layer of threshold units that each receive the same inputs to the network. It is convenient to organize the network by putting the inputs into a layer labelled "layer 0" and the threshold units into a layer labelled "layer 1". We think of the inputs as very simple "fanout" units that ramify the input signal to each of the threshold units. Denote the inputs by $x_1 \dots x_m$ and the outputs of the threshold units by $z_1, \dots, z_n$. Finally, denote the weight of the connection from input i to unit j by w_{ji}. Following the idea at the end of Section 2, we define $x_0 \equiv 1$ and let w_{j0} denote the weight of the connection from x_0 to unit j. Thus $-w_{j0}$ is the threshold of unit j. See Figure 6.1.

Vector notation is also convenient. Let $\mathbf{x} = (x_0, \dots, x_m)$ denote the vector of inputs to the network, and let $\mathbf{y} = (y_1, \dots, y_n)$ and $\mathbf{z} = (z_1, \dots, z_n)$ denote the vector of internal states and output values of the units, respectively. Let $\mathbf{w}_j = (w_{j1}, \dots, w_{jm})$ denote the vector of weights on the inputs of unit j.

The supervised learning problem can now be stated. Suppose we have an input pattern $\xi = (\xi_1, \dots, \xi_m)$ for the network and another pattern $\mathbf{I} = (I_1, \dots, I_n)$ that we would like to have for the network output. How can we increment the weights of the network in order to better achieve the association by the network? In other words, given a *computed* network output $\mathbf{z}$ and an *ideal* network output $\mathbf{I}$ for the input ξ, how can we achieve $\mathbf{z} = \mathbf{I}$?

The problem generalizes to more than one pattern. Given a number p of input patterns $\xi^1, \dots, \xi^p$ and ideal output patterns $\mathbf{I}^1, \dots, \mathbf{I}^p$, how can we move change the weights of the connections in order that $\mathbf{z}^q = \mathbf{I}^q$ for $q = 1, \dots, p$?

One Output

A simple perceptron, being in effect a collection of threshold units that each receive the same input, has units that are independent of each other. For this reason, we may as well discuss a single threshold unit to save some notational clutter. Then we can add all the extra subscripts to generalize anything we derive to the case of the simple perceptron. Thus we specialize to the case $n = 1$ and suppress the j subscripts. We also specify that the threshold units are sign units.

Figure 6.1. *A simple perceptron.*

Given input patterns ξ^q and associated ideal output values I^q, the perceptron learning rule gives a prescription for changing the weights. It specifies an incremental change for the ith weight as follows:

$$\Delta w_i^q = \begin{cases} 2\eta I^q \xi_i^q & \text{, if } z^q \neq I^q; \\ 0 & \text{, otherwise.} \end{cases} \tag{6.1}$$

The idea was based on experimental work of Hebb (1949) who found that some simple forms of learning are realized in biological synapses as increases in synaptic strength in proportion to the pre-synaptic input signal and post-synaptic output signal. The constant η is inserted to control size of the change, and the factor 2 is for convenience. η is called the *learning rate.* The weight change can be written in other convenient forms:

$$\begin{aligned} \Delta w_i^q &= \eta(1 - I^q z^q) I^q \xi_i^q \\ &= \eta(I^q - z^q)\xi_i^q. \end{aligned} \tag{6.2}$$

(These use the fact that all three variables have only ± 1 values.)

Given enough iterations of this suggested change, will the network evolve into one that satisfies $z^q = I^q$? There are cases where we know the answer must be ‘no’: if the problem $\xi^q \mapsto I^q, q = 1, \ldots, p$ is not linearly separable then a solution cannot exist. Rosenblatt (1962) and Block (1962) showed that in every case where a solution does exist, the algorithm converges:

Theorem. *If the pattern association problem is linearly separable, then the perceptron learning algorithm given by (6.2) converges to a perceptron that computes the desired association in a finite number of steps.*

We have not as yet specified how (6.2) determines an algorithm. The idea is straightforward. Loop through the patterns one at a time, presenting pattern ξ^q to the network. If the output z^q is not equal to the ideal output I^q, then immediately make a change in each weight using

$$w_i^{new} := w_i + \Delta w_i^q \tag{6.3}$$

where Δw_i^q is given by (6.1) or (6.2). If $z^q = I^q$, go on to the next pattern. Loop through the patterns, making weight changes as specified, until output is correct for each pattern. The theorem states that, if the pattern association problem has a solution, then this algorithm will find one. The unstated qualifier is that the learning rate constant must be sufficiently small. Note that, if there is one solution, there are many. See the discussion of robustness at the end of Section 5. See Rosenblatt (1962), Block (1962), Minsky and Papert (1969), or Hertz et al. (1991) for a proof.

Example. Consider the AND function. In Section 4 we discussed the problem of finding such a unit and found algrebaic methods that work. Suppose now that we want the unit to *learn* to calculate the correct output. The AND function is given by

AND		
x_1	x_2	z
$+1$	$+1$	$+1$
$+1$	-1	-1
-1	$+1$	-1
-1	-1	-1

Adding the clamped input line and pattern numbers to this information gives the enhanced table

Pattern Association Problem				
q	x_0^q	x_1^q	x_2^q	I^q
1	1	$+1$	$+1$	$+1$
2	1	$+1$	-1	-1
3	1	-1	$+1$	-1
4	1	-1	-1	-1

The algorithm initializes with random weight settings. Suppose these are (w_0, w_1, w_2) $= (1, 0, 2)$ and that $\eta = 0.15$. Applying the function $z = \text{sgn}(w_0 + w_1 x_1 + w_2 x_2)$ to the input patterns for $q = 1$ and $q = 2$ produces correct output. For $q = 3$, however, we obtain $z = +1$, where the ideal output is $I = -1$.

First Loop Through Patterns				
q	x_0^q	x_1^q	x_2^q	I^q
1	1	+1	+1	+1
2	1	+1	−1	−1
3	1	−1	+1	+1 X
4	1	−1	−1	−1

This triggers a trip through the weight change loop. In vector notation, the weight changes triggered by pattern 3 in loop 1 are (using (6.2)):

$$\begin{aligned}\Delta\mathbf{w}^3 &= .15(-1-(1))(1,-1,1)\\ &= -.3(1,-1,1)\\ &= (-.3,.3,-.3)\end{aligned} \tag{6.4.1}$$

and the resulting new weights are

$$\begin{aligned}\mathbf{w}^{new} &:= \mathbf{w} + \Delta\mathbf{w}^3\\ &= (1,0,2) + (-.3,.3,-.3)\\ &= (.7,.3,1.7).\end{aligned} \tag{6.4.2}$$

After this change, z is computed for pattern 4 and found to be correct.

Three more loops through the patterns produce similar results: The value of z for pattern 3 is incorrect, and a weight change is triggered. On the fifth loop through the patterns, it is found that the value of z is correct on patterns 1,3,4 but incorrect on pattern 2, triggering a different weight change

$$\Delta\mathbf{w}^2 = (-.3,-.3,.3) \tag{6.4.3}$$

and a corresponding new weight. At the end of the next (sixth) loop through the patterns, it is found that all values are correct.

The weight changes, new weights, and output values are summarized in Table 6.1. The threshold lines are depicted in Figure 6.2. This particular example has only one weight change triggered during each loop through the pattern set. *Not all problems will have this property.*

Table 6.1. *Weight Changes During Perceptron Learning*

Loop Number	Pattern Index	Current Weights			Weight Changes			New Weights		
1	3	1	0	2	$-.3$	.3	$-.3$	.7	.3	1.7
2	3	.7	.3	1.7	$-.3$	.3	$-.3$	.4	.6	1.4
3	3	.4	.6	1.4	$-.3$	.3	$-.3$	.1	.9	1.1
4	3	.1	.9	1.1	$-.3$	.3	$-.3$	$-.2$	1.2	.8
5	2	$-.2$	1.2	.8	$-.3$	$-.3$	.3	$-.5$	.9	1.1
6	-	$-.5$	.9	1.1	0	0	0	$-.5$	.9	1.1

Avoiding Close Calls

After a small amount of experimentation it soon becomes apparent that, while the algorithm outlined above works, the resulting sign unit may be very close to one that does not give correct output. The reason for this is that the algorithm stops as soon as the linear separator is in the correct location with respect to the input patterns in pattern space. If the last step is small, then the resulting threshold unit cannot have its seperator very far away from at least the last pattern that triggered a weight change.

In order to make the resulting threshold unit more robust, the algorithm can be enhanced to keep the separator a small distance away from each pattern. Requiring that the output be correct is equivalent to requiring that the sign of the net input be equal to the sign of the ideal output, that is, the condition $z^q = I^q$ is equivalent to the condition $y^q I^q > 0$. This last condition is one that we can modify to make the perceptron rule more robust. We require:

$$y^q I^q > r \tag{6.5}$$

for some positive number r. Thus instead of simply requiring the separator to be on the correct side of the pattern ξ^q, we require that it be on the correct side *and* outside the circle of radius r centered at ξ^q. See Figure 6.3. The improved version of perceptron learning replaces the condition $z^q \neq I^q$ in (6.1) with the negation of (6.5).

$$\Delta w_i^q = \begin{cases} 2\eta I^q \xi_i^q & \text{, if } y^q I^q \leq r; \\ 0 & \text{, otherwise.} \end{cases} \tag{6.6}$$

Using (6.6) in place of (6.1) or (6.2) in the algorithm gives the improved perceptron learning rule.

Figure 6.2. *Thresholds of units approaching the AND function.*

Multiple Outputs

The general perceptron learning algorithm (for simple, i.e., 1-layer) perceptrons is obtained by applying the procedure for one output to each unit in the network. The units being independent, the process may be applied in parallel or in any convenient order of the units.

The perceptron consists of a vector $\mathbf{x} = (x_0, x_1, \ldots, x_m)$ providing input to n sign units with output $\mathbf{z} = (z_1, \ldots, z_n)$. The internal state for unit j is y_j and the weight between input x_i and unit j is w_{ji}.

We are given a pattern association problem $\xi^q \mapsto \mathbf{I}^q, q = 1, \ldots, p$ where each ξ^q is an m-component pattern and each $\mathbf{I}^q$ is an n-component pattern. We desire to train the perceptron by incrementally changing its weights so that it reproduces the pattern association with a robust system (the separators are not too close to the input patterns). The algorithm is:

Perceptron Learning Algorithm

```
Repeat
    For $q = 1 \ldots p$ do
     1. Present $\xi^q$ to the network inputs: For $i = 1$ to $m$ do $x_i := \xi_i^q$; set $x_0 = 1$
     2. Compute the internal states $y_j$ of the threshold units: For $j = 1$ to $n$ do
        $y_j := \sum_{i=0}^{m} w_{ji} x_i$
     3. Compute the weight changes for pattern $q$: For $i = 1$ to $m$ and $j = 1$ to
        $n$, if $y_j I_j^q \leq r$ then $\Delta w_{ji}^q := 2\eta I_j^q x_j$ else $\Delta w_{ji}^q := 0$
```

Figure 6.3. *Zones of avoidance shown around input patterns.*

```
        4. Take corrective step: For $i = 1$ to $m$ and $j = 1$ to $n$ do $w_{ji}^{new} := w_{ji} + \Delta w_{ji}^{q}$
    Endfor
    Until correct and robust: $y_j I_j^q > r$ for each $j$ and each pattern $q$.
```

Theorem. *If the pattern association problem $\xi^q \mapsto \mathbf{I}^q, q = 1, \ldots, p$ is linearly separable, then for sufficiently small η and r the Perceptron Learning Algorithm converges in a finite number of iterations.*

7. Supervised Learning Algorithms.

The perceptron learning rule was introduced in 1960. Independently, and more or less simultaneously, Widrow and Hoff introduced gradient descent learning as the adaptive mechanism of adaline. These milestones introduce a general paradigm for supervised training methods in neural networks. Before proceeding with discussions of analog neurons, adaline and other developments such as backpropagation, it is helpful to standardize supervised learning algorithms.

We assume given p patterns

$$\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(q)}, \ldots, \mathbf{x}^{(p)} \tag{7.1}$$

where $\mathbf{x}^{(q)}$ has m components, i.e., $\mathbf{x}^{(q)} = (x_1^{(q)}, \ldots, x_m^{(q)})$. Assume also given ideal output patterns

$$\mathbf{I}^{(1)}, \mathbf{I}^{(2)}, \ldots, \mathbf{I}^{(q)}, \ldots, \mathbf{I}^{(p)} \tag{7.2}$$

where $\mathbf{I}^{(q)} = (I_1^{(q)}, \ldots, I_n^{(q)})$. The desired pattern association is

$$\mathbf{x}^{(q)} \mapsto \mathbf{I}^{(q)}, q = 1, 2, \ldots \tag{7.3}$$

The given pattern associations are often referred to as the *training set* for a given learning task.

A neural network with m inputs and n outputs defines a mapping (or association) from m-component patterns to n-component patterns, and thus has the potential to associate the given patterns as specified by (7.3). The perceptron rule, and other supervised learning methods, provide a basic mechanism by which the weights in a neural network are successively modified toward the ideal association given by (7.3). Each of these methods has a specific weight change computation and stopping criterion. Excepting these details, the primary learning methods that have been found useful are essentially the same.

On-Line Training

The method introduced in Section 6 with perceptron learning is called *on-line* training and is given by the following outline:

```
Repeat
    For $q = 1 \ldots p$ do
     1. Present $\mathbf{x}^{(q)}$ to the network inputs
```

```
        2. Compute the network outputs $\mathbf{z}^{(q)}$
        3. Compute the weight change $\Delta\mathbf{w}^q$ for pattern $q$
        4. Take corrective step $\mathbf{w}^{new} := \mathbf{w} + \Delta\mathbf{w}^q$
    Endfor
Until stop condition is satisfied
```

Notice that two details are required in order have a precise method: The calculation of the weight change $\Delta\mathbf{w}^q$ for pattern q and the stop condition. For the perceptron learning algorithm, these are given by (6.6) and (6.5), respectively.

This method is named for its appropriateness in situations where the incoming data stream is encountered in real time. The pattern index then represents increasing time, so that at a given instant one does not know what the next pattern will be. A training step is taken at every opportunity.

Batch Training (Off-Line Training)

When all desired pattern associations are known in advance, other methods may be more efficient. For example, it may be that the individual associations generate weight changes that are fairly large and in widely disparate directions in weight space. In such circumstances stability and speed of training may be enhanced by, in effect, averaging or summing all the weight changes suggested by the individual pattern associations and taking one corrective step after all patterns have been considered. Modifying on-line learning by extricating the weight corrective step from the inner loop results in a frequently used method called *batch training*:

```
Repeat
    For $q = 1 \ldots p$ do
        1. Present $\mathbf{x}^{(q)}$ to the network inputs
        2. Compute the network outputs $\mathbf{z}^{(q)}$
        3. Compute the weight change $\Delta\mathbf{w}^q$ for pattern $q$
    Endfor
    Accumulate weight changes $\Delta\mathbf{w} := \sum_{q=1}^{p} \Delta\mathbf{w}^q$
    Take corrective step $\mathbf{w}^{new} := \mathbf{w} + \Delta\mathbf{w}$
Until stop condition is satisfied
```

Again, all that is required to make this algorithm precise is the details of how the weight corrective step and stopping condition are defined. The cycling through p patterns in the For loop may be replaced with a random selection of p patterns from a larger pool.

Overtraining

A third possibility would be to bring the For loop outside of the Repeat loop, in effect training on one pattern until learned, then proceeding to the next pattern, and so on through the training set. This method does not work well because of the "memory instability" problem inherent in most neural networks: training on a new pattern tends to destroy previously learned associations. Thus overtraining on a fixed pattern tends to negate the effects of training on previous patterns, necessitating a third outer-most loop that again cycles through all patterns repeatedly. The net result is a training method that works, but is inordinately slow.

8. Analog Units and Graded Pattern Classification.

There are situations in which it is useful to have binary I/O for an artificial neuron, and they admit to a clean and elegant analysis elucidating their capabilities. On the other hand, neurological realism can be improved by allowing analog I/O. The analog input components can be thought of as levels of rudimentary signals, either from other neurons or from external stimuli, and the analog output is something like the average spike rate of a neuron.

We define an *analog neuron* to be a McCulloch-Pitts neuron with a continuous activation function. Analog neurons are found to be more useful in applications where some kind of graded response is appropriate, such as in neurocontrol devices or networks whose output levels are interpreted as levels of certainty, and training methods such as backpropagation use smoothness of output functions in a critical way.

Linear vs Sigmoidal Activation: Scaling the Output

As long as the activation function of an analog neuron is a strictly monotone function, such as in either the linear or sigmoidal case, the effect can be thought of as a scaling of the net input. If the output is not used as input to another neuron, this scaling effect is only one of convenience for the user: the scaling can be reversed and/or modified to any other scale. Suppose, for example, that $f(y)$ is a strictly monotone activation function and that $g(y)$ is a desired re-scaling of the net input y. Since f is monotone, it is a one-to-one mapping and as such has an inverse function f^{-1} with the property that the composition f followed by f^{-1} is the identity function:

$$(f^{-1} \circ f)(y) = f^{-1}(f(y)) = y \tag{8.8}$$

for all y. We can re-scale the output z using the function $h = g \circ f^{-1}$ with the effect that net input is scaled by g instead of f:

$$h(z) = g(f^{-1}(z)) = g(f^{-1}(f(y))) = g(y) \tag{8.9}$$

i.e., the activation function has been changed from f to g by externally manipulating the output data. In particular, we can switch back and forth between logistic and linear activation functions, or between two different sigmoidals, by re-scaling the output. (We saw in Section 2 that the logistic and hyperbolic tangent functions are related by a *linear* change of scale.)

In applications, the modeler may for reasons of convenience or realism prefer that output should be scaled to reside in some interval other than $0 \le z \le 1$ and/or should be a decreasing instead of increasing function of net input. These changes of scale can be accomplished using ideas like the ones illustrated in the previous paragraph and will be discussed as they arise in applied settings. The theory is not affected by such scale changes, so we will restrict theoretical discussions to the canonical cases described above.

Pattern Classification

In the previous section we saw that a given binary function can be realized by a binary neuron if and only if it is linearly separable. This result has an interpretation in terms of pattern classification as follows. We call a particular choice of binary inputs $(x_1, \ldots, x_n)$ an *n-dimensional binary pattern.* Then the set of all n-dimensional binary patterns is the set on which a logical function β of n variables is defined. Thus β divides the n-dimensional binary patterns into two classes, those on which β has value $+1$ and those on which β has value -1: β "classifies" the n-dimensional patterns. The classification can be realized by a sign unit if and only if β is linearly separable.

Similarly, a choice of inputs for an analog neuron can be thought of as an *n-dimensional analog pattern* $(x_1, \ldots, x_n)$, and one naturally questions what is the capacity of an analog neuron to classify such patterns. The output of an analog neuron is a real number instead of a binary value, reflecting a graded response appropriate for the continuous variation in possible input patterns. It follows that a pattern classification by an analog neuron is graded rather than binary, and such gradation must be taken into account in answering our question.

The observation that an analog neuron with strictly monotone activation function can be re-scaled to one with a linear activation function means that analog linear neurons have the same pattern classification capacity as the more general type. For convenience, then, we may restrict our investigation of pattern classification capacity to linear neurons. We return to this investigation after a short detour into linear algebra.

Vectors and the Dot Product

Concepts from linear algebra are extremely useful in both neural networks and pattern recognition. We regard $\mathbf{R}^n$ as an n-dimensional *vector space.* Points in $\mathbf{R}^n$ are then *vectors* and denoted with bold face characters, as $\mathbf{u} = (u_1, \ldots, u_n)$. A vector $\mathbf{u}$ has *coordinates* $u_1, \ldots, u_n$ and *length* or *magnitude* $|\mathbf{u}| = [(u_1)^2 + \ldots + (u_n)^2]^{1/2}$. A vector also has an absolute *direction* which is defined in terms of its direction cosines, but for our purposes we need only consider relative directions in terms of the angle between two vectors.

The *inner* or *dot product* of two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbf{R}^n$ is defined by the equation

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + \ldots + u_n v_n = \sum_{i=1}^{n} u_i v_i. \tag{8.10}$$

The similarity between equations (8.10) and (8.1) is immediately apparent and has an interpretation. We think of the weights $w_1, \ldots, w_n$ that specify an analog neuron as the coordinates of the *weight vector* $\mathbf{w}$ of the unit, and the inputs $x_1, \ldots, x_n$ for an analo neuron as the coordinates of an *input vector* $\mathbf{x}$. Then the net input is just the dot product of the weight vector and the input vector: $y = \mathbf{w} \cdot \mathbf{x}$.

Equation (8.10) gives a very handy way to compute the dot product, in effect an "algebraic" definition. There is an alternate "geometric" definition of the dot product given by

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}| \cos\theta \tag{8.11}$$

where θ is the angle between the two vectors $\mathbf{u}$ and $\mathbf{v}$. That is, the dot product is the product of the magnitudes of the vectors and the cosine of the angle between them. While (8.10) provides a convenient way of *computing* the dot product, (8.11) provides a convenient way of *interpreting* the dot product. The usefulness of the dot product stems from this dual ability to algebraically compute and geometrically interpret its value.

An example of this algebra-geometry interplay is the characterization of when two vectors are perpendicular. Suppose $\mathbf{u}$ and $\mathbf{v}$ are vectors in $\mathbf{R}^n$. Note that the two vectors are perpendicular if and only if the angle θ between them is $\pi/2$ (90 degrees), and that the cosine of $\pi/2$ is zero. In fact, $\pi/2$ is the only angle in the range $0 \leq \theta \leq \pi$ whose cosine is 0. It follows from (8.11) that $\mathbf{u}$ and $\mathbf{v}$ are perpendicular if and only if $\mathbf{u} \cdot \mathbf{v} = 0$. Applying (8.10), we have an easily checked criterion: $\mathbf{u} \perp \mathbf{v}$ *if and only if* $\Sigma u_i v_i = 0$.

The dot product is also useful in interpreting the projection of one vector along the other. Let $\mathbf{p}$ denote the projection of $\mathbf{u}$ along $\mathbf{v}$. Then, using basic trigonometry, $\mathbf{p}$ can be computed to be the vector that lies a portion $|\mathbf{u}| \cos\theta$ along the vector $\mathbf{v}$. Thus $\mathbf{p} = (|\mathbf{u}| \cos\theta)(\mathbf{v}/|\mathbf{v}|)$. Rephrasing making use of the dot product equation (8.11), we have: *The projection* $\mathbf{p}$ *of* $\mathbf{u}$ *along* $\mathbf{v}$ *is given by the equation*

$$\mathbf{p} = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{v}|^2}\mathbf{v}. \tag{8.12}$$

In the special case where $\mathbf{v}$ has magnitude 1, $\mathbf{p} = (\mathbf{u} \cdot \mathbf{v})\mathbf{v}$. See Figure 8.2a.

Graded Pattern Classification

Return now to the question of how a linear analog neuron classifies input patterns. We assume the unit has n-dimensional weight vector $\mathbf{w}$ and receives an n-dimensional input vector (pattern) $\mathbf{x}$. The output is $z = \mathbf{w} \cdot \mathbf{x} = |\mathbf{w}||\mathbf{x}| \cos\theta$. To simplify the notation, let us assume that the weight vector has magnitude 1 (this assumption does not restrict the pattern classification capacity of the neuron) and let us also restrict the input patterns to those of magnitude 1 (the magnitude of a pattern is a measure *a.* $\mathbf{p}$ = *projection of* $\mathbf{u}$ *along* $\mathbf{v}$.

b. The four vectors in Example 1.

Figure 8.2.

of its overall intensity, so we are simply restricting our consideration to patterns of the same overall intensity = 1). Under these assumptions, we have

$$z = \cos\theta \tag{8.13}$$

where θ is the angle between $\mathbf{w}$ and $\mathbf{x}$.

The angle θ is restricted to the interval $0 \le \theta \le \pi$, the endpoints representing the two extreme cases ($\theta = 0$ means the vectors point in the same direction and $\theta = \pi$ means they point in opposite directions). Thus the cosine of θ varies from +1 to −1 and uniquely determines the angle θ. Since by assumption $\mathbf{w}$ and $\mathbf{x}$ have the same magnitude, the two extreme cases are $\cos\theta = +1$, meaning that $\mathbf{x} = \mathbf{w}$, and $\cos\theta = -1$, meaning that $\mathbf{x} = -\mathbf{w}$. The intermediate case is $\cos\theta = 0$, meaning that $\mathbf{x} \perp \mathbf{w}$. Thus the unit classifies patterns according to how well they match the internal pattern stored as its weight vector. See Figure 8.2.

Example 1. Let $\mathbf{w} = (0.9000, -0.4359)$. Then

$$\begin{aligned} |\mathbf{w}| &= [(0.9000)^2 + (-0.4359)^2]^{1/2} \\ &= [0.8100 + 0.1900]^{1/2} \\ &= [1.0000]^{1/2} = 1 \end{aligned} \tag{8.14.1}$$

so $\mathbf{w}$ defines a linear unit with 2 inputs and weight vector of magnitude 1. Consider three input vectors

$$\mathbf{x}^{(1)} = (0.8500, -0.5268)$$

$$\begin{aligned} \mathbf{x}^{(2)} &= (-0.8800, 0.4750) \\ \mathbf{x}^{(3)} &= (0.4300, 0.9028). \end{aligned} \tag{8.14.2}$$

It should be checked that each of these patterns has magnitude 1 and that the various dot products with $\mathbf{w}$ are as given in (8.14.3) below:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}^{(1)} &= 0.9946 \\ \mathbf{w} \cdot \mathbf{x}^{(2)} &= -0.9991 \\ \mathbf{w} \cdot \mathbf{x}^{(3)} &= -0.0065. \end{aligned} \tag{8.14.3}$$

These output values $z^{(p)} = \mathbf{w} \cdot \mathbf{x}^{(p)}, p = 1 \ldots 3$, may be interpreted as follows. $z^{(1)}$ is close to 1 so $\mathbf{x}^{(1)}$ is nearly the same pattern as $\mathbf{w}$; $z^{(2)}$ is close to -1 so $\mathbf{x}^{(2)}$ is nearly the opposite of $\mathbf{w}$; and $z^{(3)}$ is close to 0 so $\mathbf{x}^{(3)}$ has nearly nothing in common with $\mathbf{w}$. See Figure 8.2b. ▮

We depart this section with a cautionary note. The figures one draws to illustrate these concepts are usually two-dimensional, as in Figure 8.2, thus representing the case $n = 2$. There is one aspect of such pictures that is misleading: the size of the subspace orthogonal to $\mathbf{w}$. In general, the weight vector $\mathbf{w}$ defines a 1-dimensional subspace of pattern space $\mathbf{R}^n$, which means that the subspace orthogonal to $\mathbf{w}$ is $(n-1)$-dimensional. A single linaer unit is forced to have the value $z = 0$ for all input vectors $\mathbf{x}$ that lie in that orthogonal subspace. A single analog neuron thus has nothing to say about most of the input patterns.

9. Adaline.

This section introduces a method for training or adapting networks of analog units. The method was introduced by Widrow and Hoff (1962) around the same time as Rosenblatt brought forth the perceptron learning method. Widrow and Hoff called the method the "delta rule" or "LMS" algorithm (for least mean squares). LMS is an example of what is now called *gradient descent learning*, or GDL. We will cover background theory and applications in later sections. Here we concentrate on describing the algorithm, writing prototype code, and suggesting some experiments in supervised learning.

Adaline

Widrow and Hoff coined the term "Adaline" from the phrase "ADAptive LINear Element" for an analog unit equipped with LMS learning. Consider a single linear unit. Thus we have an *input vector* $\mathbf{x} = (x_1, \ldots, x_m)$, a *weight vector* $\mathbf{w} = (w_1, \ldots, w_m)$, and *output* equal to net input given by the dot product

$$y = \mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{m} w_i x_i. \tag{9.1}$$

Suppose we are given a pattern $\mathbf{x}^{(0)}$ and an ideal output $I^{(0)}$ for the unit. To get the unit to perform correctly, that is, to have the desired output on $\mathbf{x}^{(0)}$, we could use linear algebra techniques to calculate values for the weights that would work. An alternative approach is to allow the unit to adapt its own weights over time toward a weight vector that gives the desired behavior. In this alternate approach we consider the unit to adapt or learn by changing its internal state (its weights) rather than the more traditional view of finding a unit that has the correct knowledge. A linear unit, together with this adaptation algorithm, is called an "Adaline" (for adaptive linear neuron).

The adaptation procedure goes as follows. First present Adaline with the input pattern $\mathbf{x}^{(0)}$ and compute Adaline's output $y^{(0)}$. Next compare this result with the ideal output $I^{(0)}$. The *error* is given by

$$e = I^{(0)} - y^{(0)}. \tag{9.2}$$

If the error is zero then Adaline has the correct behavior. Otherwise we change weights in an attempt to reduce the magnitude of error and try again.

The error reduction process is built around an attempt to to minimize *square error*

$$E = e^2. \tag{9.3}$$

Note that minimizing square error results in zero error, whereas (since error may be negative) it is not appropriate to attempt to minimize error itself. The iterative modification of Adaline's weights takes a "step" in the weight space. The step is given by one of the formulae

$$\Delta \mathbf{w} = \frac{\eta e}{|\mathbf{x}^{(0)}|^2}\mathbf{x}^{(0)} \tag{9.4.1}$$

$$\Delta \mathbf{w} = \frac{\eta e}{|\mathbf{x}^{(0)}|}\mathbf{x}^{(0)} \tag{9.4.2}$$

$$\Delta \mathbf{w} = \eta e \mathbf{x}^{(0)} \tag{9.4.3}$$

where $|\mathbf{x}^{(0)}|$ is the magnitude of $\mathbf{x}^{(0)}$, e is error, and η is a learning rate parameter to be set externally. As is becoming usual, the step given by (9.4) is then added to the existing weights before commencing a new trial:

$$\mathbf{w}^{new} = \mathbf{w} + \Delta \mathbf{w}. \tag{9.5}$$

Note that the recipe (9.4.3) is identical in form to the perceptron update rule (6.2). We will derive each of these in the next section using criteria completely different from the Hebb rule. The three update rules above differ only in the scalar denominator, so they each take a step in the same direction in weight space. When the input vector $\mathbf{x}^{(0)}$ is not extremely long or short, all three of these recipes give essentially the same results. The distinction among them is how they react to wide ranges of input vector sizes.

Formalizing the procedure described above results in the following supervised learning algorithm for training a single unit on a single pattern $\mathbf{x}^{(0)}$:

```
Repeat
  1. Present $\mathbf{x}^{(0)}$ to the unit inputs;
  2. Compute the output $z^{(0)}$ using (9.1);
  3. Compute the error $e$ using (9.2);
  4. Compute the weight change $\Delta\mathbf{w}$ using (9.4);
  5. Take corrective step $\mathbf{w}^{new}$ using (9.5);
Until $E$ is small.
```

A depiction of Adaline is given in Figure 9.1. Values of the weights at the beginning of this procedure are assumed given. These initial weight values are usually either selected at random or input from some previous training experience. A multiple output Adaline network consists of a number of Adaline elements (one for each independent output value) receiving the same input vector in parallel and is trained by simply carrying out the procedure above for each element.

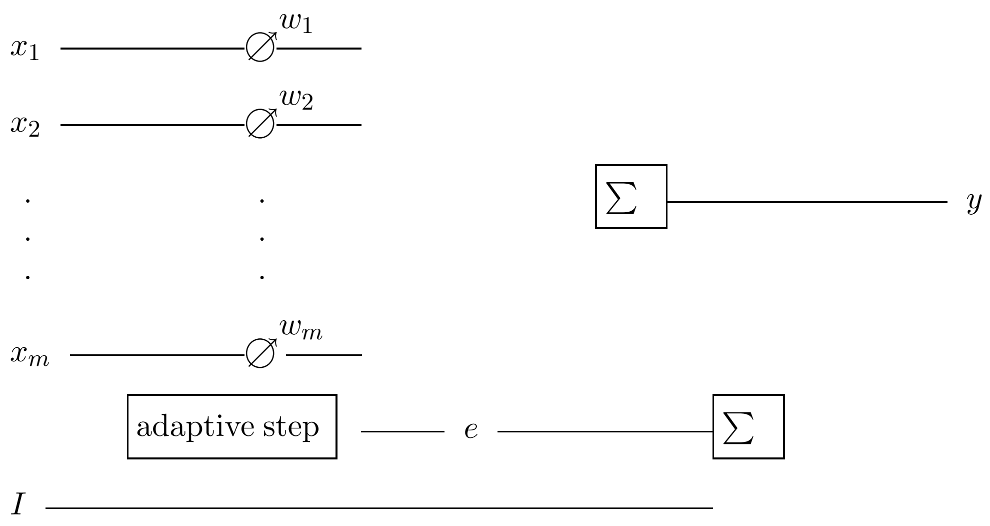

Figure 9.1. *Adaline.*

Programming Adaline

We now present generic computer code that implements the Adaline learning algorithm discussed above. The term "generic" means two things: (1) the code is readily translated into actual computer code written in a declarative modular language such as FORTRAN, Pascal, and C; and (2) the generic code is comprehensible and self-explanatory when read carefully by humans. These two properties, *translate-ability* and *readability*, are important characteristics of generic code. We will build on the code introduced here in later sections.

It is highly recomended that the reader take time now to encode a rapid prototype version of Adaline. We suggest setting up the program so that it receives matrices as input arrays, allowing interpretation as patterns in two spatial dimensions. (Note that this does not mean 2-dimensional patterns: The dimension of a pattern is the number of degrees of freedom in the pattern. Thus the patterns discussed in the experiments below are 25-dimensional.) For readers who want to avoid coding, however, source code for Adaline is available free through electronic mail. In the experiments suggested below, we assume that the input matrix is read into the input array in column-major order, i.e., by reading the first row, then the second row, and so on, until the entire matrix is read.

Adaline Code

The first function predict uses Adaline in its current state, as defined by a set of weight values, to compute a value for the input pattern. The activation function phi is assumed programmed somewhere. Of course, phi may be the identity function (the strict linear case) which requires no program. We suggest either letting phi be a logistic function with threshold equal to zero and gain equal to four or simply letting phi be the identity. In any case, the learning process is independent of phi.

```
function predict
import  m {number of inputs}
        n {number of outputs}
        x[i], i=1..m {input to network}
        w[j,i], j=1..n, i=1..m {weights}
export  z[j], j=1..n {output from network}
begin
  for j = 1 to n do
    y[j] ← 0 {initialize}
    for i = 1 to m do
      y[j] ← y[j] + w[j,i]*x[i]
    endfor {dot product loop}
    z[j] ← phi(y[j]) {apply activation function}
  endfor
end
```

The next procedure correct modifies the weights of the network using a given set of errors between ideal and actual output values.

```
procedure correct
import  m {number of inputs}
        n {number of outputs}
        x[i], i=1..m {input to network}
        z[j], j=1..n {output from network}
        e[j], j=1..n {error in output from network}
        eta {learning rate}
effect  w[j,i], j=1..n, i=1..m {weights}
begin
  mag_squared ← 0
  for i = 1 to m do
    mag_squared ← mag_squared +x[i]*x[i]
  endfor
  for j = 1 to n do
    for i = 1 to m do
```

```
      delta_w[j,i] ← eta *e[j]*x[i]/mag_squared
    endfor
  endfor
  for j = 1 to n do
    for i = 1 to m do
      w[j,i] ← w[j,i] + delta_w[j,i]
    endfor
  endfor
end
```

The learning process itself consists of iteratively making prediction and correction until a desired level of accuracy is reached or the loop times out. Procedure `learn` calls `predict` and `correct` as subroutines.

```
procedure learn
{implements single pattern learning for Adaline network}
import  epsilon {desired level of accuracy}
        maxits {maximum iterations allowed}
        eta {learning rate}
        m {number of inputs}
        n {number of outputs}
        x[i], i=1..m {input to network}
        I[j], j=1..n {ideal network output}
        w[j,i], j=1..n, i=1..m {initial weights}
export  stop_condition {minimized, timed out}
effect  w[j,i], j=1..n, i=1..m {change state of weights}
begin
  predict
  for j = 1 to n do {compute errors}
    e[j] ← I[j] - z[j]
  endfor
  square_error ← 0
  for j = 1 to n do {compute square error}
    square_error ← square_error + e[j]*e[j]
  i ← 0
  while
    square_error > epsilon
      and i < maxits
  do {learnung loop}
    correct
    predict
    for j = 1 to n do {compute errors}
      e[j] ← I[j] - z[j]
    endfor
    square_error ← 0
    for j = 1 to n do {compute square error}
      square_error ← square_error + e[j]*e[j]
    endfor
    i ← i + 1 {count iteration}
  endwhile
  if square_error <= epsilon
    then stop_condition ← minimized
    else stop_condition ← timed_out
  endif
end
```

Adaline can be used in two modes. In learning mode, weights are modified to change the outcomes of predictions. After learning to recognize appropriate input patterns, the net can then be used to evaluate any other pattern using the function `predict` by itself, with learning "turned off". Learning to associate more than one pattern simultaneously requires building another loop (through the pattern training set) that implements on- line or batch learning as discussed in Section 7.

Experimenting With Adaline

Once having Adaline code available, experimentation is inevitable, instructive, and fun. We recommend some specific experiments for a single Adaline (i.e., an Adaline network with one output). These can be modified, with similar results, for a multiple output net. The activation function is assumed to be the identity. η is assumed to lie in the interval $0 \leq \eta \leq 1$ with $\eta \approx 0.1$ recommended. The suggested experiments refer to the following five input patterns.

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \; P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \; X = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \tilde{F} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & \epsilon \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Experiment 1. (Basic training.) Learn to recognize each of the patterns F, P, and X by associating $F \to 1$, $P \to 2$, and $X \to 3$. Begin with a randomly selected set of weights in each case. Look at the weights before and after training. Are any predictable things happening? ▮

Experiment 2a. (Learning more than one pattern.) Teach Adaline to associate all three patterns simultaneously. Try different techniques for instruction, including on-line, batch, and "overtraining". ▮

Experiment 2b. Repeat Experiment 2a with the three associations $F \to 1$, $P \to 2$, and $U \to 3$. Note any difficulties you may have as compared to Experiment 2a and attempt to explain them. ▮

Experiment 3. (Generalization.) Train Adaline to associate patterns $F \to 1$ and $U \to 2$ simultaneously. What is the predicted value for P? What about the pattern $\tilde{F}$ where ϵ is a number representing corruption? Try the same with corruption of F in a different place. ▮

Results of these experiments can be explained using the theory in Section 11 and the observation that F, P, and X are linearly independent patterns while F, P, and U satisfy the linear equation $F + U = P$.

10. Gradients and Steepest Descent.

There are three related topics covered in this section. First we review concepts surrounding the gradient of a function of several variables. Next we apply these concepts to derive the damped gradient descent method for minimizing such a function. We conclude with a generalization to mappings appropriate for neural net applications, the same method used in the earlier introduction to supervised learning (Adaline).

The Gradient

Suppose that f is a real-valued continuously differentiable function defined on some open set in euclidean n-space $\mathbf{R}^n$. If $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$ is a point (vector) in the domain of f and $\mathbf{u} = (u_1, \ldots, u_n)$ is a unit vector, the *directional derivative* of f at $\mathbf{x}^{(0)}$ in the direction $\mathbf{u}$ is given by

$$D_{\mathbf{u}} f(\mathbf{x}^{(0)}) = \left[\frac{d}{ds} f(\mathbf{x}^{(0)} + s\mathbf{u})\right]_{s=0}. \tag{10.1}$$

Consider the variable $\mathbf{x}^{(0)} + s\mathbf{u}$ for a moment. As the real parameter s moves from $-\infty$ to $+\infty$, this variable parametrizes the line through the point $\mathbf{x}^{(0)}$ with $\mathbf{u}$ as its positive direction. Therefore the function $s \longrightarrow (\mathbf{x}^{(0)} + s\mathbf{u}, f(\mathbf{x}^{(0)} + s\mathbf{u}))$ parametrizes the curve on the graph of f that lies over this line. The derivative in (10.1) is then the slope of this curve. The derivative at $s = 0$ is the slope of the curve at the point where it is directly over $\mathbf{x}^{(0)}$. Thus, the directional derivative is the slope of the line that is tangent to the graph of f over the point $\mathbf{x}^{(0)}$ in the $\mathbf{u}$ direction.

There are several facts that are useful in understanding directional derivatives. The first says that reversing the direction of $\mathbf{u}$ results in changing the sign of the directional derivative; it is easily verified by direct substitution into (10.1):

$$D_{-\mathbf{u}} f(\mathbf{x}^{(0)}) = -D_{\mathbf{u}} f(\mathbf{x}^{(0)}). \tag{10.2}$$

The second is obtained by applying the chain rule to the derivative in equation (10.1) as follows:

$$\begin{aligned}\frac{d}{ds} f(\mathbf{x}^{(0)} + s\mathbf{u}) &= \sum_i \frac{\partial f}{\partial x_i}(\mathbf{x}^{(0)} + s\mathbf{u}) \frac{d}{ds}(x_i^{(0)} + su_i) \\ &= \sum_i \frac{\partial f}{\partial x_i}(\mathbf{x}^{(0)} + s\mathbf{u}) u_i.\end{aligned}$$

Taking the limit as $s \to 0$ we obtain

$$D_{\mathbf{u}} f(\mathbf{x}^{(0)}) = \sum_i \frac{\partial f}{\partial x_i}(\mathbf{x}^{(0)}) u_i \tag{10.3}$$

which is the basis of the following

Definition. The *gradient* of f at the point $\mathbf{x}^{(0)}$ is the vector

$$\nabla f(\mathbf{x}^{(0)}) = (\frac{\partial f}{\partial x_1}(\mathbf{x}^{(0)}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}^{(0)})). \tag{10.4}$$

The directional derivative can then be expressed as a dot product of the gradient and the direction vector:

$$D_{\mathbf{u}} f(\mathbf{x}^{(0)}) = \nabla f(\mathbf{x}^{(0)}) \cdot \mathbf{u} \tag{10.5}$$

which follows by direct application of (2.8), (10.3) and (10.4). ▮

Recall that the dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$ has the alternate definition $\mathbf{u}{\cdot}\mathbf{v} = |\mathbf{u}||\mathbf{v}| \cos\theta$, where θ is the angle between $\mathbf{u}$ and $\mathbf{v}$ (see equation (2.9)). Applying this result to equation (10.5), we see that the directional derivative is maximized when $\cos\theta = 1$, i.e., when $\theta = 0$. Thus, $D_{\mathbf{u}} f(\mathbf{x}^{(0)})$ is maximized when $\mathbf{u}$ and $\nabla f(\mathbf{x}^{(0)})$ have the same direction. This proves the first part of the following fundamental property of the gradient.

Theorem 1. *The gradient of a function f at a point $\mathbf{x}^{(0)}$ is the vector whose direction is in the direction of maximum increase of f at $\mathbf{x}^{(0)}$ and whose magnitude is that maximal rate of increase.*

Proof. The direction part has been proved above. For the second part we calculate the directional derivative (using equation (10.5)) in the direction of the gradient, that is, we take $\mathbf{u} = \nabla f(\mathbf{x}^{(0)})/|\nabla f(\mathbf{x}^{(0)})|$:

$$\begin{aligned} D_{\mathbf{u}} f(\mathbf{x}^{(0)}) &= \nabla f(\mathbf{x}^{(0)}) \cdot \mathbf{u} \\ &= \nabla f(\mathbf{x}^{(0)}) \cdot \frac{\nabla f(\mathbf{x}^{(0)})}{|\nabla f(\mathbf{x}^{(0)})|} \\ &= \frac{|\nabla f(\mathbf{x}^{(0)})|^2}{|\nabla f(\mathbf{x}^{(0)})|} \\ &= |\nabla f(\mathbf{x}^{(0)})| \end{aligned}$$

which proves the magnitude assertion.

If we think of the graph of f as a landscape over the ground plane $\mathbf{R}^n$, then Theorem 1 may be interpreted as saying that the gradient vector at a given point

Figure 10.1. *Gradient Descent. When standing on the ground plane at* $\mathbf{x}^{(0)}$*, the negative gradient at* $\mathbf{x}^{(0)}$ *points the way to decrease* $f(\mathbf{x})$ *most rapidly.*

$\mathbf{x}^{(0)}$ (which is a vector in the ground plane) is always pointing in the direction of steepest ascent available at $\mathbf{x}^{(0)}$. ∎

Similarly we can observe that the directional derivative at $\mathbf{x}^{(0)}$ is minimized when $\theta = \pi$ or $\cos\theta = -1$, and the argument proceeds in a similar manner to show:

Theorem 2. *The negative gradient* $-\nabla f(\mathbf{x}^{(0)})$ *points in the direction of maximum rate of decrease of* f *at* $\mathbf{x}^{(0)}$ *and has magnitude equal to this maximum rate of decrease.*

(Keep in mind that a positive rate of decrease is a negative rate of increase.) Again using the landscape analogy, we would say that the negative gradient points in the direction of steepest descent of f. The topic of gradient descent is discussed further in the later section Numerical Considerations. ∎

Gradient Descent Learning

Suppose we have a single analog neuron with weight vector $\mathbf{w} = (w_1, \ldots, w_m)$, strictly monotone activation function φ, input vector $\mathbf{x} = (x_1, \ldots, x_m)$, net input $y = \mathbf{w} \cdot \mathbf{x}$, and output or response $z = \varphi(y)$.

In training the neuron, we consider both $\mathbf{x}$ and $\mathbf{w}$ to be variable; variation in $\mathbf{x}$ represents variation in patterns presented to the unit, while variation in $\mathbf{w}$ represents changing or adapting the unit in order to obtain a desired response to one or more input patterns. After training, the weight vector $\mathbf{w}$ is held constant and the response z is determined as before from variable input $\mathbf{x}$. In order to train the unit to produce a "correct" response to a given input pattern $\mathbf{x}^{(0)}$, we assume that an ideal response $I^{(0)}$ is given. Thus we desire to train the unit so that $z^{(0)} = I^{(0)}$. The Widrow-Hoff "delta rule" method of training the unit uses gradient descent applied to the square error function with the goal of reducing square error to zero.

At this point, purely for convenience and without any loss of generality, we assume that the unit is linear, i.e., that $\varphi(y) = y$. The general case is recovered by simply replacing the ideal output with its inverse $\varphi^{-1}(I^{(0)})$ under φ. (See the discussion Scaling the Output in Section 8.) Thus $z = y = \mathbf{w} \cdot \mathbf{x}$.

Define the error associated with the situation described above by the equation

$$e = I^{(0)} - z^{(0)} = I^{(0)} - \sum_{i=1}^{n} w_i x_i^{(0)}. \tag{10.6}$$

We want to minimize square error $E = e^2$ using gradient descent, where E is considered as a function of w and input is held constant. Applying the chain rule and (10.6), the gradient of E is calculated as

$$\begin{aligned} \nabla E &= 2e\nabla e \\ &= 2e(\frac{\partial e}{\partial w_1}, \dots, \frac{\partial e}{\partial w_n}) \\ &= 2e(-x_1^{(0)}, \dots, -x_n^{(0)}) \\ &= -2e\mathbf{x}^{(0)}. \end{aligned} \tag{10.7}$$

Therefore the negative gradient has direction equal to the unit vector $\pm(\mathbf{x}^{(0)}/|\mathbf{x}^{(0)}|)$ with the sign given by $\operatorname{sgn}(e)$. This completes a proof of the following

Theorem 3. *The direction of fastest decrease in square error $E = e^2$ for an analog neuron is plus or minus the direction of the input $\mathbf{x}^{(0)}$, the sign being the same as the sign of the error e.*

It remains to derive an algorithm for modifying the weights. The algorithm will consist of moving one step a certain distance in the direction indicated by Theorem 3. The size of the step should be dependent on the relative error (the ratio of magnitude of error and magnitude of input), giving a small step when error is small. Taking relative error as the step length yields the following computation of the step vector:

$$\begin{aligned} \Big[\text{length}\Big]\Big[\text{direction}\Big] &= \left[\frac{|e|}{|\mathbf{x}^{(0)}|}\right]\left[\operatorname{sgn}(e)\frac{\mathbf{x}^{(0)}}{|\mathbf{x}^{(0)}|}\right] \\ &= e\frac{\mathbf{x}^{(0)}}{|\mathbf{x}^{(0)}|^2}. \end{aligned} \tag{10.8}$$

The gradient descent learning (GDL) update rule is obtained by inserting a learning rate constant η into (10.8) as a damping factor:

$$\begin{aligned}\Delta\mathbf{w} &= \mathbf{w}^{new} - \mathbf{w}\\ &= \eta e \frac{\mathbf{x}^{(0)}}{|\mathbf{x}^{(0)}|^2}.\end{aligned} \tag{10.9.1}$$

The learning rate parameter η is discussed further below.

Equation (10.9.1) is derived using relative error. If absolute error were used instead, we would obtain

$$\Delta\mathbf{w} = \eta e \frac{\mathbf{x}^{(0)}}{|\mathbf{x}^{(0)}|} \tag{10.9.2}$$

and if we simply use the gradient itself we obtain

$$\Delta\mathbf{w} = \eta e \mathbf{x}^{(0)}. \tag{10.9.3}$$

Note that these are the three update rules given in (9.4).

The Widrow-Hoff Algorithm

The Widrow-Hoff GDL algorithm is a ramification of the gradient descent method to a network consisting of a number of analog neurons acting in parallel on the same input vector. It is strikingly similar in form to the perceptron learning rule. Both GDL and perceptron learning were derived around 1960, but they use completely different bases for derivation and were discovered independently (Rosenblatt, 1962; Widrow and Hoff, 1962). The perceptron update rule came from Hebb's biological experiments. The GDL update rule comes from the geometry of the square error surface.

Thus we have an input vector $\mathbf{x} = (x_1, \ldots, x_m)$ whose components fan out to each of p units. The j^{th} unit has input vector $\mathbf{x}$, weight vector $\mathbf{w}_j = (w_{j,1}, \ldots, w_{j,m})$, net input y_j, activation function f_j, and output z_j, as described previously in Section 9. The Widrow-Hoff algorithm applies the delta rule to each unit in parallel fashion. Again and for the same reasons, we assume the units are linear, so that $z_j = y_j$ for each j.

When presented with a particular input $\mathbf{x}^{(0)}$ for which we have an ideal output $I_j^{(0)}$ for the j^{th} unit, we can define the error as

$$e_j = I_j^{(0)} - z_j^{(0)} = I_j^{(0)} - \sum_{i=1}^{m} w_{ji} x_i^{(0)} \tag{10.10}$$

and the *total square error* as

$$E = (e_1)^2 + \ldots + (e_p)^2. \tag{10.11}$$

Then applying equation (10.9.1) to each unit separately yields the delta rule

$$\Delta \mathbf{w}_j = \eta e_j \frac{\mathbf{x}^{(0)}}{|\mathbf{x}^{(0)}|^2} \tag{10.12}$$

for each $j = 1 \ldots p$. Restating (10.12) in scalar form gives

$$\Delta w_{ji} = \eta e_j \frac{x_i^{(0)}}{\sum_{k=1}^{m} (x_k^{(0)})^2}. \tag{10.13}$$

The delta rule (10.13) applies to update the weights using

$$w_{ji}^{new} = w_{ji} + \Delta w_{ji}. \tag{10.14}$$

Supervised learning for a single pattern $\mathbf{x}^{(0)}$ then proceeds as follows.

1. Present the network with the (non-zero) pattern $\mathbf{x}^{(0)}$;
2. Compute the network outputs $z_1^{(0)}, \ldots, z_n^{(0)}$ as in Section 8;
3. Compute the errors and total square error using (10.10) and (10.11);
4. If total square error is small go to step 7;
5. Modify the weights according to (10.13) and (10.14);
6. Return to step 1;
7. Stop.

This is the learning procedure that was used in the experimental discussions of Adaline in Section 9.

The damping constant η, often referred to as the "learning rate", is a parameter that facilitates some external control on step size and is assumed to be in the range $0 \leq \eta \leq 1$; typically η is smaller than 1/2. Another external parameter, a kind of "artificial mass" denoted here by μ, has also been found useful in the form of a "momentum term" added to the right side of (10.13), yielding a modified delta rule

$$\Delta w_{ji}^{new} = \eta e_j \frac{x_i^{(0)}}{\sum_{k=1}^{m} (x_k^{(0)})^2} + \mu \Delta w_{ji}. \tag{10.15}$$

Momentum is discussed further in a later section. The problem of learning multiple associations is taken up in the next section.

Stepsize Normalization

The factor $|\mathbf{x}^{(0)}|^2$ in the denominator of (10.12) is a scalar and hence does not effect the direction of weight change, just the stepsize. As long as the training pattern is not very small or very large in magnitude, this normalizer has little effect. We have argued that in theory it is beneficial especially when the pattern is large. In practice, often this may be omitted, however, simplifying the calculation. See the discussion of (10.9) above.

11. General Theory of Linear Associators.

The theory of linear associators (Adaline networks) is quite complete and comprehensible and provides an excellent microcosm in which to examine several questions related to more general neural networks. These questions are: What is the capacity of a network to learn more than one task? What is the capacity to generalize to unlearned but similar tasks? And how does the gradient descent method work on the problem of multiple learning? These questions are answered precisely in the context of Adaline. The answers may provide intuitive guidelines for more complex networks.

Matrix Operations and The Vector Box Product

Matrices provide an extremely useful notational language for dealing with large quantities of vector information and linear transformations of such information. We review here those concepts and notations that are essential for our discussion of linear associators. We begin with some definitions.

Matrices. A $p \times n$ *matrix* is a rectangular array $A = (a_{ji})$ of real numbers with p rows and n columns:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \dots & a_{pn} \end{pmatrix}. \tag{11.1}$$

A is said to have *dimension* $p \times n$. p is equal to the number of rows, or dimension of each column, and is called the *column dimension* of A. n is equal to the number of columns, or dimension of each row, and is called the *row dimension* of A. The quantity a_{ji} is called the (ji) *entry* of A; this entry occurs at the intersection of the j^{th} row and the i^{th} column of A. The various rows and columns of a matrix define vectors. For example, the first column of A above defines the vector $(a_{11}, a_{21}, \dots, a_{p1})$ and the j^{th} row of A defines the vector $(a_{j1}, a_{j2}, \dots, a_{jn})$. Often the distinction between the row or column of A and the vector it defines is blurred. Conversely, a vector $\mathbf{v} = (v_1, \dots, v_n)$ can be thought of as a $1 \times n$ matrix. Several operations are defined on matrices (of appropriate dimension) including matrix sum, scalar multiplication, matrix product, and matrix transpose.

Matrix Addition. Given two $p \times n$ matrices $A = (a_{ji})$ and $B = (b_{ji})$, the matrix sum is defined by adding the individual entries:

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} + b_{p1} & a_{p2} + b_{p2} & \dots & a_{pn} + b_{pn} \end{pmatrix}. \tag{11.2}$$

Addition is not defined for matrices not of the same dimension. Note that if $\mathbf{u}$ and $\mathbf{v}$ are vectors then the vector sum $\mathbf{u}+\mathbf{v}$ is identical with the matrix sum $\mathbf{u}+\mathbf{v}$.

Scalar Multiplication. If A is a matrix and α is a scalar, the scalar product is defined by mutliplying each entry of A by the scalar α:

$$\alpha A = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} & \dots & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & \dots & \alpha a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha a_{p1} & \alpha a_{p2} & \dots & \alpha a_{pn} \end{pmatrix}. \tag{11.3}$$

Again, the product of a scalar and a vector is the same as the product of the scalar and the matrix defined by the vector.

The Matrix Product. Suppose that $A = (a_{kj})$ and $B = (b_{ji})$ are matrices of dimension $p \times n$ and $n \times m$, respectively. The product of A and B is defined to be the matrix $C = (c_{ki})$ whose (ki) entry is the dot product of the k^{th} row of A with the i^{th} column of B:

$$c_{ki} = \sum_{j=1}^{n} a_{kj} b_{ji}. \tag{11.4}$$

For example,

$$\begin{aligned} &\begin{pmatrix} 1 & 2 & -1 \\ 3 & 0 & -2 \end{pmatrix} \begin{pmatrix} 1 & -2 \\ 0 & 4 \\ -1 & -1 \end{pmatrix} \\ &\qquad = \begin{pmatrix} row_1 \cdot col_1 & row_1 \cdot col_2 \\ row_2 \cdot col_1 & row_2 \cdot col_2 \end{pmatrix} \\ &\qquad = \begin{pmatrix} 1\times 1 + 2\times 0 + (-1)\times(-1) & 1\times(-2) + 2\times 4 + (-1)\times(-1) \\ 3\times 1 + 0\times 0 + (-2)\times(-1) & 3\times(-2) + 0\times 4 + (-2)\times(-1) \end{pmatrix} \\ &\qquad = \begin{pmatrix} 2 & 7 \\ 5 & -4 \end{pmatrix}. \end{aligned}$$

Note that the row dimension of A (second dimension factor) must be equal to the column dimension of B (first dimension factor) in order for the definition to make sense, and that the dimension of the product is $p \times m$.

The Matrix Transpose. Let $A = (a_{ji})$ be a $p \times n$ matrix. The transpose of A, denoted by $A^T = (a^T_{ji})$, is obtained by "flipping" A about its main diagonal:

$$a^T_{ji} = a_{ij}. \tag{11.5}$$

Two special cases of transpose matrices are of interest. Normally a vector is written in the form $\mathbf{v} = (v_1, \dots, v_n)$ so that it is in the form of a $1 \times n$ matrix. To distinguish this case a $1 \times n$ matrix is often called a *row vector*. An $n \times 1$ matrix consists of a single column of n numbers and obviously contains exactly the same information as a $1 \times n$ matrix. An $n \times 1$ matrix is often referred to as a *column vector*. The transpose operation switches back and forth from one vector form to the other:

$$(v_1\ v_2 \dots v_n)^T = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{11.6.1}$$

and

$$\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}^T = (v_1\ v_2 \dots v_n). \tag{11.6.2}$$

-2

The Vector Box Product. Suppose that $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$ are (row) vectors. We have defined and used to advantage the *inner* or *dot product* (see Section 10). The dot product formula (10.8) can be restated in terms of matrix operations as follows:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}\mathbf{v}^T \tag{11.7}$$

That is, the dot product of $\mathbf{u}$ and $\mathbf{v}$ is the unique entry of the 1×1 matrix obtained as the matrix product of the row vector $\mathbf{u}$ and the column vector $\mathbf{v}^T$. Another useful vector product is obtained by a small modification of (11.7). The *outer* or *box product* of the vectors $\mathbf{u}$ and $\mathbf{v}$ is given by the equation

$$\mathbf{u} \,\Box\, \mathbf{v} = \mathbf{u}^T\mathbf{v} \qquad (11.8).$$

Note that the distinction between (11.7) and (11.8) is in which vector is transposed. The small change in notation makes a large change in the outcome, however: in the case of dot product, we multiply matrices of dimension $(1 \times n)(n \times 1)$ obtaining a 1×1 matrix (a single number). In the case of box product, we multiply matrices of dimension $(n \times 1)(1 \times n)$ obtaining an $n \times n$ matrix (n^2 numbers).

For example, let $\mathbf{u} = (1, 2, 3)$ and $\mathbf{v} = (1, -2, 2)$. Then

$$\mathbf{u} \cdot \mathbf{v} = (\,1 \quad 2 \quad 3\,) \begin{pmatrix} 1 \\ -2 \\ 2 \end{pmatrix} = 3$$

and

$$\mathbf{u} \,\square\, \mathbf{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 1 & -2 & 2 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 2 \\ 2 & -4 & 4 \\ 3 & -6 & 6 \end{pmatrix}.$$

The box product is a square array consisting of all possible products of a coordinate of $\mathbf{u}$ with a coordinate of $\mathbf{v}$.

Linear Associators

Equipped now with the powerful language of matrix and vector products, we can give a comprehensible explanation of the general case of linear pattern classification begun in Section 10. We start with the case of a single input pattern (assumed non-zero, i.e., of some positive intensity).

One Pattern. The point of view is that we are given a pattern $\mathbf{x}^{(0)}$ together with a set of n desired or "ideal" responses $I_j^{(0)}, j = 1 \ldots n$, and we seek n linear units that associate the input patterns with the n ideal outputs. The j^{th} linear unit we seek has input vector $\mathbf{x}$ and computes its output value $z_j = \mathbf{w}_j \cdot \mathbf{x}$, where $\mathbf{w}_j$ is the weight vector of the unit. We think of the input pattern $\mathbf{x}^{(0)}$ and the ideal outputs $I_j^{(0)}$ as given and the weight vectors as unknowns to be sought. If we find correct weights, the resulting units collectively associate the input pattern with the desired output; they form a linear associator.

The notation can be simplified considerably by grouping the output values z_j into a single *output vector* $\mathbf{z} = (z_1, \ldots, z_n)$ and considering the weight vectors $\mathbf{w}_j$ as the rows of a *weight matrix* W. The linear associator can then be described in terms of matrix operations as

$$\mathbf{z}^T = W\mathbf{x}^T. \tag{11.9}$$

(Recall that the transpose just writes vectors in column form.) Therefore the *linear associator problem* is to find weights $W = (w_{ji})$ so that if $\mathbf{z}^{(0)T} = W\mathbf{x}^{(0)T}$ then $\mathbf{z}^{(0)} = \mathbf{I}^{(0)}$, where $\mathbf{I}^{(0)}$ is the vector of ideal outputs for the pattern.

The linear associator problem for a single pattern can be solved exactly using dot products. We seek (for each j) a weight vector $\mathbf{w}_j$ for which $\mathbf{w}_j \cdot \mathbf{x}^{(0)} = I_j^{(0)}$. Using equation (10.9) we write the dot product as $\mathbf{w}_j \cdot \mathbf{x}^{(0)} = |\mathbf{w}_j||\mathbf{x}^{(0)}| \cos\theta$ where θ is the angle between $\mathbf{w}_j$ and $\mathbf{x}^{(0)}$. Interpreting this last equation trigonometrically, we conclude that we seek a vector $\mathbf{w}_j$ which projects a distance $I_j^{(0)}/|\mathbf{x}^{(0)}|$ along the vector $\mathbf{x}^{(0)}$. (See Figure 11.1.) There are two important observations to make about this constraint on $\mathbf{w}_j$.

Figure 11.1. *The hyperplane H of solutions to the single pattern associator problem.*

Observation 1. *The vector* $\mathbf{w}_j = (I_j^{(0)}/|\mathbf{x}^{(0)}|^2)\mathbf{x}^{(0)}$ *works.*

This observation is verified by direct computation:

$$\begin{aligned}\mathbf{w}_j \cdot \mathbf{x}^{(0)} &= \left(\frac{I_j^{(0)}}{|\mathbf{x}^{(0)}|^2}\mathbf{x}^{(0)}\right) \cdot \mathbf{x}^{(0)} \\ &= \frac{I_j^{(0)}}{|\mathbf{x}^{(0)}|^2}\left(\mathbf{x}^{(0)} \cdot \mathbf{x}^{(0)}\right) \\ &= I_j^{(0)} \end{aligned} \tag{11.10}$$

Observation 2. *There are many other vectors that also work.*

This follows from the geometric fact that if a vector lies in the hyperplane perpendicular to $\mathbf{x}^{(0)}$ passing through the point of projection then the vector will also project to the same point. (See Figure 11.1. This hyperplane is important when we discuss gradient descent learning.)

We can now write down an exact solution to the linear associator problem for one pattern. Let $W = (w_{ji})$ be the matrix defined by

$$w_{ji} = \frac{I_j^{(0)} x_i^{(0)}}{|\mathbf{x}^{(0)}|^2} \tag{11.11}$$

(where $i = 1, \ldots, m$ and $j = 1, \ldots, n$). Then the calculation in (11.10) above shows that

$$W\mathbf{x}^{(0)T} = \mathbf{I}^{(0)T} \tag{11.12}$$

which is the condition required by (11.9). Thus, in particular, there is no need for an iterative learning algorithm in this case, we can just compute a solution using (11.11).

A final observation is appropriate as we move to the case of learning multiple patterns. Suppose that the pattern $\mathbf{x}^{(0)}$ has magnitude equal to one. (This simply means we assume $\mathbf{x}^{(0)}$ has unit intensity.) Then the formula given by (11.11) defining the weight matrix W reduces to the box product:

$$W = \mathbf{I}^{(0)} \,\square\, \mathbf{x}^{(0)}. \tag{11.13}$$

Orthonormal Patterns. Now suppose we are given a number p of input patterns $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(p)}$ along with ideal output vectors $\mathbf{I}^{(1)}, \ldots, \mathbf{I}^{(p)}$ and we desire to associate output $\mathbf{I}^{(k)}$ with $\mathbf{x}^{(k)}$ for each k using a single Adaline network. That is, we seek a single weight matrix $W = (w_{ji})$ such that

$$W\mathbf{x}^{(k)T} = \mathbf{I}^{(k)T} \tag{11.14}$$

holds for each $k = 1 \ldots p$. This is the linear associator problem for multiple patterns.

We begin with a set of input patterns that is *orthonormal*, that is, we assume that the input patterns (1) have unit length and (2) are mutually perpendicular. The first of these assumptions is just a simplification to unit intensity, but the second implies a constrained geometry that is unrealistic. We shall remove these constraints later in this section, but they provide an essential link in the theory.

The assumption of orthonormality can be phrased in terms of the dot product as

$$\mathbf{x}^{(j)} \cdot \mathbf{x}^{(i)} = \delta_{ji} \tag{11.15}$$

where δ_{ji} is the Kronecker delta function (equal to 1 when $j = i$ and equal to 0 when $j \neq i$). One consequence of orthonormality is that the set of input patterns is linearly independent and, in particular, $p \leq n$.

Using the observation of (11.13), we define a weight matrix

$$W^{(k)} = \mathbf{I}^{(k)} \,\square\, \mathbf{x}^{(k)} \tag{11.16}$$

for each k. We know from the discussion above that the matrix $W^{(k)}$ associates the pattern $\mathbf{x}^{(k)}$ with the ideal output $\mathbf{I}^{(k)}$. Now define W to be the matrix sum

$$W = W^{(1)} + W^{(2)} + \ldots + W^{(p)} \tag{11.17}$$

and check how W associates, say, the l^{th} pattern:

$$\begin{aligned} W\mathbf{x}^{(l)T} &= \left(\sum_k W^{(k)}\right)\mathbf{x}^{(l)T} \\ &= \left(\sum_k \mathbf{I}^{(k)} \,\square\, \mathbf{x}^{(k)}\right)\mathbf{x}^{(l)T} \\ &= \sum_k \left(\mathbf{I}^{(k)T}\mathbf{x}^{(k)}\right)\mathbf{x}^{(l)T} \\ &= \sum_k \mathbf{I}^{(k)T}\left(\mathbf{x}^{(k)}\mathbf{x}^{(l)T}\right) \\ &= \sum_k \mathbf{I}^{(k)T}\left(\mathbf{x}^{(k)} \cdot \mathbf{x}^{(l)}\right) \\ &= \mathbf{I}^{(l)T}. \end{aligned} \tag{11.18}$$

That is, the association is perfect! Again, we have simply written down a formula for the weight matrix W and there is no need for an iterative learning scheme. This is the point at which the ability to "write down" the solution breaks down, however. Notice that the assumption of orthonormality plays a critical role in the computation of (11.18). In the last line, all interaction in the sum disappears because of our assumption that (11.15) holds. Without orthonormality, the calculation would be hopelessly complicated.

Independent Patterns. Now replace the assumption of orthonormality by the assumption of linear independence. This is a realistic constraint on a set of input patterns (and an essential one in the case of linear associators). The theory is affected significantly. It is still possible to prove the existence of a solution to the associator problem, but there is no longer a simple method of computing what the solution is. We examine both of these effects. The proofs are somewhat intuitive but require some background in linear algebra to follow them completely.

Claim 1. *There is a solution to the linear associator problem for linearly independent patterns.*

A proof of this claim goes as follows. Assume the input patterns $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(p)}$ are linearly independent. Then we can expand them to a basis for $\mathbf{R}^m$ and then map this basis to an orthonormal basis via a linear transformation. This linear transformation has a matrix A which must satisfy: *The patterns $A\mathbf{x}^{(1)T}, A\mathbf{x}^{(2)T}, \ldots, A\mathbf{x}^{(p)T}$ form an orthonormal set.* Therefore, using the results above, we can find a weight

Figure 11.2. *The error surface for a linear associator problem is a parabolic trough whose bottom is the hyperplane of solutions.*

matrix $\tilde{W}$ such that $\tilde{W}(A\mathbf{x}^{(k)T}) = I^{(k)T}$ for $k = 1, \ldots, p$. If we define W to be the product matrix $\tilde{W}A$, it follows that $W\mathbf{x}^{(k)T} = I^{(k)T}$ for each k, proving the claim. ▮

The problem now is how to compute W. One approach would be to find a matrix A making the input patterns orthonormal. This is a computationally expensive task (order $O(p^2 n)$) and implementationally complex. Another approach is to use iterative methods such as gradient descent. These can be much better than the direct linear algebra approach. The gradient descent update rule, in particular, is nice in that it generalizes directly to the case of non-linear multilayer networks.

Claim 2. *Gradient descent learning will always converge to a solution to the linear associator problem for a collection of linearly independent input patterns.*

A proof of Claim 2 emerges from the following general discussion of gradient descent.

Remarks On Gradient Descent Learning

Recall from our discussion in Section 10 that gradient descent update rule applied to the j^{th} unit establishes a corrective step for the j^{th} weight vector. If the input pattern is $\mathbf{x}^{(0)}$ and the ideal output is $I_j^{(0)}$, this corrective step has direction equal to plus or minus the direction of $\mathbf{x}^{(0)}$, the sign being the sign of error $e_j = I_j^{(0)} - z_j^{(0)}$. The magnitude of the step is proportional to the magnitude of relative error.

This corrective step is obtained from the gradient of the square error $E_j = e_j^2$ as a function of $\mathbf{w}_j$. Due to the quadratic nature of E_j, its graph is a "parabolic trough", a trough with parabolic cross-section whose "bottom" is the $(m-1)$-dimensional hyperplane of correct solutions to the associator problem depicted in Figure 11.1. The direction of the corrective step is perpendicular to, and toward, the trough bottom. The magnitude of the corrective step is proportional to the distance away from the bottom. It will be helpful to give this hyperplane, the trough bottom, a name. It depends on the input pattern and the output node, so we name it $H_j^{(0)}$ after each. $H_j^{(0)}$ is perpendicular to $\mathbf{x}^{(0)}$ and consists of all solutions $\mathbf{w}$ to the associator problem defined by $\mathbf{x}^{(0)}$ and $I_j^{(0)}$. The hyperplane and the line through $\mathbf{x}^{(0)}$ intersect at the vector mentioned in Observation 1 above. See Figure 11.2.

Thus gradient descent update rule is an iteration that produces a path of steps from any point in weight space directly to the hyperplane of correct solutions. The particular correct solution so obtained depends on the particular weight used to initiate the process. Now consider the case of multiple input patterns $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(p)}$ (assumed linearly independent), with ideal output vectors $\mathbf{I}^{(1)}, \mathbf{I}^{(2)}, \ldots, \mathbf{I}^{(p)}$. Gradient descent learning may be applied to each of these desired associations. For the k^{th} input pattern and the j^{th} output node, we obtain a parabolic trough with bottom $H_j^{(k)}$ as above.

Observation 3. *Because these hyperplanes are perpendicular to vectors (the input patterns) that are linearly independent, the hyperplanes have a non-void intersection. Any weight in this intersection will satisfy all of the associator problems simultaneously.*†

If GDL update rule is applied for each of the patterns in some order, the result will be a path toward a point in the intersection. The nature of the path is dependent on the order in which the various GDL updates are applied, but convergence to a solution is guaranteed in any case (assuming the damping factor is small enough to keep the iteration from blowing up). Two such paths are depicted in Figure 11.3 for the case of two patterns. These paths correspond to two different training schemes. The first alternates between patterns until a (simultaneous) solution is found; the second trains on pattern 1 until learned, then trains on pattern 2 until learned, then returns to pattern 1, repeating until converged to a simultaneous solution.

Conclusions

We return to the questions raised at the beginning of the section. **Q:** What is the capacity of Adaline to learn multiple patterns? **A:** An m-input Adaline can learn up to m linearly independent patterns.

† This provides a geometric proof of Claim 1. It also shows why the sum in (11.18) works: when the patterns are orthogonal, the sum of the weight vectors (rows of W), when calculated by the parallelogram rule, ends up in the intersection of the trough bottoms, because they are mutually perpendicular.

Figure 11.3. *Paths defined by applying GDL for two different patterns in different ways.*

Q: How well does the gradient descent method work for learning multiple patterns? **A:** Quite well in the linear case. Some researchers have reported best performance begins to drop off when the number p of training patterns excedes 30–40% of capacity (i.e., when $p \sim .4n$) but there is no reason in principle why $p = n$ is unattainable. Depending on the angles involved in the intersecting trough bottoms in weight space, some variation in choice of path (learning scheme) and convergence constants (learning rate and artificial mass) may be helpful.

Q: What is the capacity of Adaline to generalize to unlearned patterns? **A:** Adaline uses linear interpolation to generalize. For example, if she has the value .99 on the pattern $\mathbf{x}^{(1)}$ and the value .01 on the pattern $\mathbf{x}^{(2)}$ then, for a pattern that is 95% $\mathbf{x}^{(1)}$ and 5% $\mathbf{x}^{(2)}$ she has the value $.95 \times .99 + .05 \times .01 = .941$. Note, however, that if Adaline has not been trained on a complete set of n independent patterns then her values will be non-sensical for any pattern with a component that is outside the space spanned by the training set. Thus, for example, if Adaline has been trained on $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ but not $\mathbf{x}^{(3)}$, then for a pattern that is 5% $\mathbf{x}^{(3)}$ Adaline's prediction may be nonsense since it may still be influenced by the random choices made to initialize GDL in training on the first two patterns. ∎

What about the non-linear case? If the network topology remains as considered here, that is, a single layer of analog units with no feedback, the capacity of the network is unchanged. The addition of logistic activation functions cannot add any new ability not already possessed by the linear version. (See the remarks on scaling the output in Section 10.) As we shall soon see, the situation changes dramatically when we expand to a multilayer topology.

12. Case Study: Adaptive Filters.

Some of the earliest and most successful applications of neural network technology fall in the area of adaptive signal processing using Adaline networks. This is pioneering work fathered by Bernard Widrow and dating back to the early 1960's. It is in this early work that the name "Adaline" was coined, standing for "ADAptive LInear NEuron" or "ADAptive LINear Element" (depending on whether artificial neural networks were in or out of fashion at the time). The term was first applied to threshold units but has expanded its scope over the years to include various analog units. Thus *Adaline* refers to a single McCulloch-Pitts neuron together with the Widrow-Hoff supervised learning algorithm.

There are now many examples of the effective use of Adaline networks in adaptive control or modification of signals. A review of some of these was recently given by Widrow and Winter. (See Widrow and Winter (1988) or Widrow and Stearns (1985).) We describe the generic *adaptive filter* and give three illustrations of its use.

Adaptive Filters

Consider the adaptive filter whose configuration is depicted in Figure 12.1. A digital signal (time-dependent bit stream) s is sampled in time by the application of delays in the signal. Thus at a given instant t, $s(t)$ is the current signal, $s(t-1)$ is the signal one delay unit in the past, $s(t-2)$ is the signal two delay units in the past, and so on, until $s(t-T)$ is the signal T delay units in the past. The output y of the filter is the dot product of the weight vector $\mathbf{w} = (w_0, \ldots, w_T)$ and the time-sampled signal vector $\mathbf{s}(t) = (s(t), \ldots, s(t-T))$:

$$y(t) = \mathbf{w} \cdot \mathbf{s}(t). \tag{12.1}$$

Clearly this is structurally equivalent to a linear analog neuron with $(T+1)$ inputs. The output signal consists of the input signal as "filtered" according to the values of the weights. By varying the weights, the output signal is directly controllable.

The filter of Figure 12.1 is called *adaptive* because the weights are modified over time so as to reduce the error between the output signal and some ideal signal. In particular, we assume that the Widrow-Hoff gradient descent learning rule is used to reduce square error. Thus an adaptive filter is one manifestation of an Adaline.

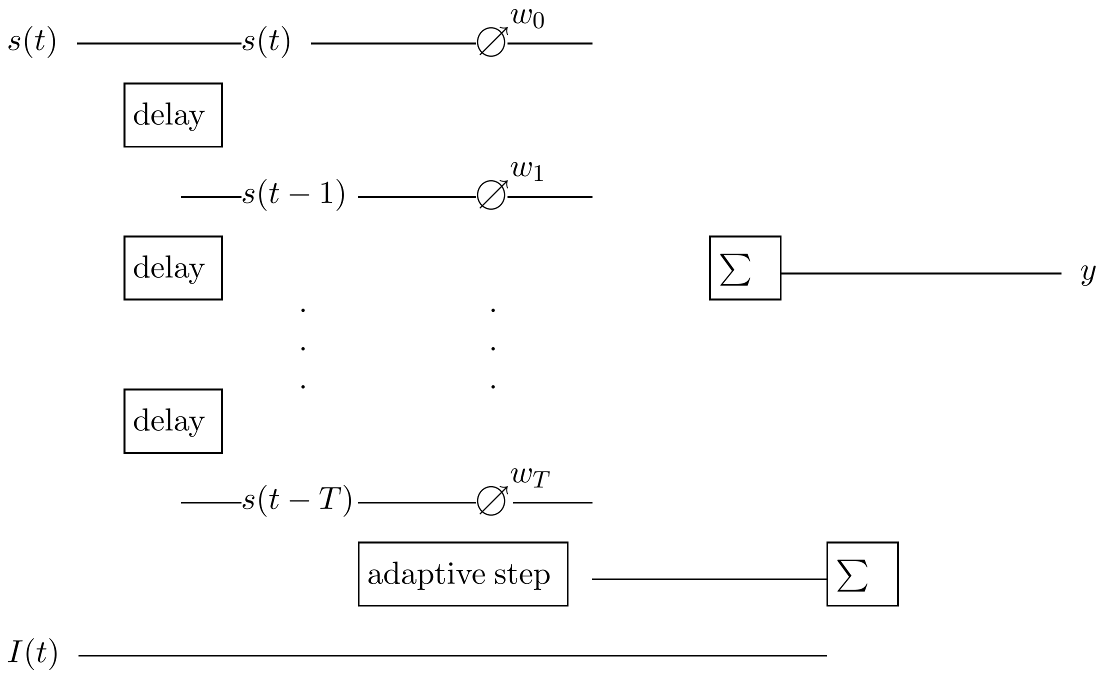

Figure 12.1. *Adaptive Filter.*

The Fourier Transform

The discrete *Fourier transform* of a time-sampled digital signal

$$\mathbf{s}(t) = (s(t), \ldots, s(t-T)) \tag{12.2}$$

breaks $\mathbf{s}$ into its periodic components. Each of these components is defined by an amplitude and frequency. The result $\hat{\mathbf{s}}$ of applying the Fourier transform to $\mathbf{s}$ can be considered a vector indexed by frequency:

$$\hat{\mathbf{s}} = (\hat{s}_1, \hat{s}_2, \ldots) \tag{12.3}$$

where $\hat{s}_k$ is the amplitude of the periodic component ϕ_k with frequency k. (In practice, frequency higher than some value may be ignored, so a "truncated" Fourier transform vector contains all information necessary to reconstruct the signal.) The *inverse* Fourier transform just sums the pure periodic components, recovering the original signal:

$$s(t) = \sum_k \hat{s}_k \phi_k(t). \tag{12.4}$$

The classical idea behind correcting a noisy signal is to break the signal apart into its periodic components using the Fourier transform, identify the frequencies associated with the noise, and omit these frequencies when the transform is inverted, obtaining a purified signal. Because the Fourier transform and the adaptive filter operate on the same time-sampled signal, the filter has all the information necessary to directly modify the frequency structure of the signal.

Figure 12.2. *Adaptive Noise Canceller.*

Noise Cancelling

The typical communications channel (microwave, copper wire, fiber optic) transmits signals consisting of a sum of sinusoidal functions that may be thought of as direct analogs of components of sound. (These are the Fourier components of the signal.) Thus "pure" sound is represented by a single component while a complex sound may have many components. If the channel is analog, we assume that digital-to-analog conversion is made immediately before sending and that analog-to-digital conversion is made immediately on receipt; all pre- and post-processing of the signal is done on the digital version. Interpretation of a received signal is complicated considerably by the fact of life that communication channels are not perfect media: they are subject to noise, interference, and other degradation from both external and internal sources. External noise is usually a complex sound ("white" noise refers to a uniform mix of Fourier components) while internal degradation is often more predictable in character, given channel characteristics such as medium and length. (These are not characteristics that are constant, even during the course of a single telephone connection.)

Consider a channel with transmitted signal s and noise n_0. The received signal is represented by the sum $s + n_0$. The objective is to recover s, in real time of course. The classical non-adaptive method of reducing noise is to "filter" the noise components using some version of the Fourier analysis alluded to above, detecting the periodic components of noise and omitting them from the reconstructed signal. The adaptive method of Widrow uses "cancelling" of the noise components. Noise cancelling is an additive process in which the negative of the noise is added to the

Figure 12.3. *Adaptive Echo Canceller.*

noisy signal effectively cancelling the noise components exactly, thus (received signal) + ($-$noise) $= (s + n_0) + (-n_0) = s$. The problem is that we do not know either s or n_0.

A device for adaptive noise cancelling is depicted in Figure 12.2. This device uses a known reference noise n_1 as input to an adaptive filter whose output y is added to the corrupted signal, producing $e = s + n_0 + y$ as the final output of the device. This output e is used as the "error" in the adaptive algorithm. Because both noises n_0 and n_1 are assumed uncorrelated with the uncorrupted signal s, the output y of the filter cannot reduce the strength of s. Thus, square error cannot minimize below the value s^2. When this minimum is attained, that is, when full adaptation has occurred, we have

$$e = s + n_0 + y = s \tag{12.5}$$

which means that the adaptive filter has learned how to convert the noise n_1 into the negative of the noise n_0 and that the output of the noise canceller is the original uncorrupted signal s. Widrow goes on to show that if s is uncorrelated with both n_0 and n_1 and if n_0 and n_1 are correlated then the adaptation algorithm converges as expected. These methods work extremely well when the general range of noise is somewhat predictable and in different frequency ranges from the transmitted signal.

Echo Cancellation

Echo in telephone channels is created by the use of two circuits, one for each direction of the two-way communication. Two circuits are required because the one-way signal must be amplified to overcome power loss during transmission. The two

Figure 12.4. *Adaptive Channel Equalizer.*

one-way circuits consititute a closed directed loop. Echo is caused by an incoming signal bleeding across the terminal transformer and re-propogating as an outgoing signal on the other half of the loop. The delay is normal transmission delay in long-distance circuits.

A pair of adaptive filters, bypassing each transformer, can be used to effectively cancel echo as indicated in Figure 12.3. The input for the filter is the incoming signal, the output is subtracted from the outgoing signal, and this output plays the role of error. The convergence of the filter to a best adaptive state is guaranteed by the same theory as described for the noise canceller above. Square error cannot minimize to zero but only to the power of the uncorrupted outgoing signal, so after adaptation the error signal consists of the outgoing signal with echo cancelled. Use of adaptive echo cancellation in long-distance telephone circuits is increasing rapidly.

Channel Equalization

The basic idea behind sending binary information along a channel is to convert the bit stream into a pulsed signal, where each bit is represented by a pulse. The actual signal may be analog or digital, but the pulses represent discrete information superimposed on the signal. If the channel itself is digital, it is helpful to keep in mind that the information pulse is superimposed macroscopically on the digital stream. The dual send/receive functions of a computer "modem" (MODulator-DEModulator) are (1) to convert an outgoing bit stream to a signal for transmission and (2) to convert an incoming transmitted signal to a bit stream. (There is digital-to-analog conversion at the end of step (1) and analog-to-digital conversion at the beginning of step 2. The signal is superimposed on a pure tone carrier wave that is subtracted away upon receipt.) The first of these tasks is relatively straightforward, since the only information required is a specification of the channel. The second is complicated by inevitable corruption during transmission.

A form of corruption particularly detrimental to binary information is intersymbol interference, caused by a degradation of the pulse in the time direction, a phenomenon called "smearing". Smearing can result in error rates as high as 10% in recovering binary data from telephone channels. An adaptive channel equalizer, which uses an idea of R.W. Lucky called "decision-directed" learning together with an adaptive filter and a quantizer, can reduce error rates to 10^{-6} or less. (See Lucky (1965).) Adaptive equalization circuits are virtually universal components in today's data modems.

An adaptive equalizer is depicted in Figure 12.4. One of the central weights in the filter is initialized to 1 and the others to zero, a state which would allow an unsmeared signal to pass through unchanged. The output of the quantizer would be the transmitted pulse. With smearing the quantized output may at first differ significantly from the input to the quantizer, causing adaptation to reduce the error. Once adapted, the quantized output is the original quantized signal uncorrupted by smearing.

BIBLIOGRAPHY

Aho, A. V.; Hopcroft, J. E.; and Ullman, J. D. (1983). *Data Structures and Algorithms*, Addison-Wesley, Reading, MA.

Alkon, D. L.; ...; T. P. Vogl (1990). Pattern-recognition by an artificial network derived from biological neuronal systems. *Biological Cybenetics* **62**, 363-376.

Anderson, J. A.; Rosenfeld, E. (1988). *Neurocomputing: Foundations of Research.* MIT Press, Cambridge, MA. (A collection of many important papers.)

Block, H. D. (1962). The perceptron: A model for brain functioning. *Reviews of Modern Physics* **34**, 123-135.

Durens, A. J.; Filkin, D. L. (1989). Efficient training of the back-prop network by solving a system of stiff ordinary differential equations. *IJCNN 89*, vol.II, 381-386.

Fahlman, S. E.; Lebiere, C. (1990). The cascade correlation learning architecture. CMU Technical Report CMU-CS-90-100, Carnegie Mellon University School of Computer Science, Pittsburgh.

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks* **2** (3), 183-92.

Gelinas, R. J. (1972). Stiff systems of kinetic equations – a practitioner's view. *J. of Computational Physics* **9**, 222- 236.

Gleick, J. (1987). *Chaos.* Viking, New York.

Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science* **11**, 23-63.

Hecht-Nielsen, R. (1987). Kolmogorov's mapping theorem. *Proc. IEEE Int. Conf. Neural Networks.*

Hebb, D. O. (1949). *The Organization of Behavior.* Wiley, New York.

Hertz, J.; Krogh, A.; Palmer R.G. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley, New York.

Hirsch, M. W., (1989). Convergent activation dynamics in continuous time networks. *Neural Networks* **2**, 331-349.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA* **79**, 2554-2558.

Hornik, K. M.; Stinchcombe, M.; White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* **2** (5), 359-66.

Hruska, S. I.; Dalke, A. P.; Ferguson, J. J.; Lacher, R. C. (1991). Expert Networks in CLIPS. *Proceedings 2nd Annual CLIPS Conference* (J. Giarratano and C. Culbert, eds.), NASA Scientific and Technical Information Branch, Houston, 1991, pp 267-272.

Hruska, S. I.; Kuncicky, D. C.; Lacher, R. C. (1991). Resuscitation of certainty factors in expert networks. *Proceedings IJCNN 91 - Singapore*, IEEE 91CH3065-0, November, 1991, pp 1653-1657.

Hruska, S. I.; Kuncicky, D. C.; Lacher, R. C. (1992). Hybrid learning in expert networks. *Proceedings IJCNN 91 - Seattle* (vol. II), IEEE 91CH3049-4, July, 1991, pp 117-120.

Kohonen, T. (1974). An adaptive memory principle. *IEEE Transactions on Computers* **C-23**, 444-445.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybenetics* **43**, 59-69.

Kolen, J. F.; Pollack, J. B. (1990). Back propagation is sensitive to initial conditions. *Complex Systems* **4**, 269-280.

Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Dokl. Akad. Nauk. USSR* **114**, 953-6.

Kuncicky, D. C.; Hruska, S. I.; Lacher, R. C. (1992). The equivalence of expert system and neural network inference. *International Journal of Expert Systems* **4** (3), 281-297.

Lacher, R. C. (1992a). Node error assignment in expert networks. In *Hybrid Architectures for Intelligent Systems* (A. Kandel and G. Langholz, ed.), CRC Press, New York, pp 29-48.

Lacher, R. C. (1992b). Expert networks: Paradigmatic conflict, technological rapprochement. *Minds and Machines* **2**, 19pp.

Lacher, R. C. (1992c). The symbolic/sub-symbolic interface: hierarchical network organizations for reasoning. *AAAI 92 Workshop on Integrating Neural and Symbolic Processes*, AAAI, July, 1992, pp 17-22.

Lacher, R. C.; Kuncicky, D. C.; Hruska, S. I. (1992). Backpropagation learning in expert networks. *IEEE Transactions on Neural Netwroks* **3** (1), 62-71.

Lucky, R. W. (1965). Automatic equalization for digital communication. *Bell Systems Technology Journal* **44**, 547- 88.

Manausa, M. E.; Lacher, R. C. (1991). Parameter sensitivity in the backpropagation learning algorithm. *Proceedings IJCNN 91 - Singapore*, IEEE 91CH3065-0, November, 1991, pp 390-395.

McCulloch, W. S; Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bull. Math. Biophysics* **5**, 115-133.

Meade, C. (1989). *Analog VLSI and Neural Systems.* Addison- Wesley, Reading, MA.

Minsky, M.; Papert, S. (1969). *Perceptrons.* MIT Press, Cambridge, MA. (Second edition 1988).

Moody, J.; Darken, C. (1989). Fast learning in networks with locally-tuned processing units. *Neural Computation* **1**, 281-294.

Newell, A. (1990). *Unified Theories of Cognition* (1987 William James Lectures). Harvard University Press, Cambridge, MA.

Nguyen, D.; Widrow, B. (1989). The truck backer-upper: an example of self-learning in neural networks. *IJCNN 89*, vol.II, 357-63.

Pao, Y.-H. (1989). *Adaptive Pattern Recognition and Neural Networks.* Addison-Wesley, New York.

PDP (1986). See Rumelhart and McClelleand (1986).

Pritchard, W. S.; Duke, D. W. (1990). Dimensional analysis of the human electroencephalogram using the Grassberger-Procaccia method, Technical Report FSU-SCRI-90-115, Supercomputer Computations Research Institute, Florida State University.

Rosenblatt, F. (1962). *Principles of Neurodynamics.* Spartan Press, New York.

Rumelhart, D. E.; McClelland, J. L. (1986). *Parallel Distributed Processing.* MIT Press, Cambridge, MA. (Also called PDP (1986)).

Seidenberg, M. S.; McClelland, J. L. (1989). A distributed developmental model of word recognition and naming. *Psychological Review* **96**, 523-68.

Sejnowski, T. J.; Rosenberg, C. R. (1987). Parallel networks that pronounce English text. *Complex Systems* **1**, 145-68.

Simpson, P. K. (1990). *Artificial Neural Systems*, Pergamon Press, New York, 1990.

Skarda, C. A.; Freeman, W. (1987). How brains make chaos in order to make sense of the world. *Behavior and Brain Sciences* **10**, 161-195.

Stinchcombe, M.; White, H. (1989). Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions. *IJCNN 89*, vol. I, 613-17.

Werbos, P.J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD Thesis, Harvard University.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks* **1**, 339-356.

Widrow, B. (1962). Generalization and information storage in networks of adaline "neurons". *Self-Organizing Systems 1962*, M. C. Yovits, G. T. Jacobi, and G. D. Goldstein, eds., Spartan, Washington, DC, 435-461.

Widrow, B.; Hoff, M. E. (1960). Adaptive switching circuits. *1960 IRE WESTCON Conv. Record, Part 4*, 96-104.

Widrow, B.; Winter, R. (1988). Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition. *IEEE Computer* **21**, 25- 39.

Widrow, B.; Stearns, S. D. (1985). *Adaptive Signal Processing.* Prentice Hall, Englewood Cliffs, NJ.

Ziegler, U. M.; Hawkes, L. M.; Lacher, R. C. (1992). Rapid learning with large weight changes and plasticity. *Proceedings IJCNN 92 - Baltimore* IEEE 92CH3114-6, Vol. II, June, 1992, pp 146-151.

æ