Chapter 8

Artificial Neural Networks

History: The last 50 years has produced a revolution in mankind's notion of the nature of intelligence. Powerful computational machines have been developed capable of handling vast amounts of information at incredible rates, and methodologies have been developed that produce machine experts rivaling the best expertise that mankind has to offer. Yet with all this development, there remain many classes of computation which have not yet yielded to these knowledge intensive, symbolic processing approaches. Sensorimotor tasks appear to be more difficult computational problems than playing chess or solving problems in differential calculus. We have had tremendous success with computerized chess and calculus, and relatively little when it comes to processing visual information, generating skillfull sensorimotor behavior, generalizing problem solving to similar domains, and learning from experience.

Partly as a result of this disparity, and from dramatic progress in neuroscience, it seems reasonable to consider the effect of architecture on the feasibility of computation associated with sensorimotor behavior. In this chapter, we will consider the aspects of *computing by connection.* We will outline on a history of the field that touches on some of the major developments.

1943 *McCulloch-Pitts Neuron*

1949 *Hebbian Learning Rule*

1957 *Rosenblatt's Perceptron Convergence Rule*

1969 *Minsky and Papert's "Perceptrons" Paper*

1982 *Hopfield Networks*

1986 *Backpropagation*

Computability In this chapter, we will develop a model for computation based on a network of McCulloch-Pitts *neurons* that is equivalent in expressive power to the classical Turing model. This result demonstrates that every computable problem could in principle be solved by a network of simple processor elements. The McCulloch-Pitts model was devised as a sort of first order simulation of the networks of neurons in the brain.

Learning The experiences of an animal change its nervous system over time. The pattern of electrical activity leaves *traces* in the network of cells, changing the state of the brain and adapting the way in which it processes information. These changes are broadly referred to as learning (as distinguished from other forms of behavioral changes resulting from fatigue or trauma). Learning involves many different kinds of behavior modification such as conditioned reflexes (Pavlov), the ability to remember a face or a song, and the acquisition of motor skill. *Learning* usually refers to the process of acquiring new information; whereas *memory* refers to the persistence of what is learned.

In this chapter, we will discuss the implications of "computing through connectivity" as a mechanism for learning and adaptation. We will introduce the correlational synapse and associative memory in order to motivate the Hopfield network which can be taught to distinguish patterns in its input information. The perceptron is capable of classifying inputs into disjoint sets using the delta rule learning algorithm. This procedure only requires instances of correct classifications which the perceptron generalizes to novel inputs. This learning architecture is significantly enhanced by multiple layer networks which propagate classification errors backward through network layers from output to input.

8.1 The McCulloch-Pitts Model

Section 2.2.1 discussed the structure of the multipolar neuron, in which a tree-like structure of inputs (excitatory/inhibitory) synapse with the cell body. The output of this type of cell consists of a single axon which could project some distance from the cell to pass a depolarization event onto a multitude of other cells. The McCulloch-Pitts neuron is a computational simulation of the biological neuron. Figure 8.1 is a schematic of the MP neuron.

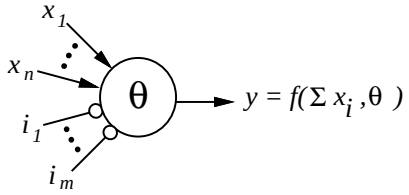


Figure 8.1 *The McCulloch-Pitts Neuron Model*

The MP neuron model has n input signals, $x_j,\ j = 1, n$ and m inhibitory inputs, $i_j,\ j = 1, m$. The

output depends on the cell activation threshold, θ, and the output *squashing* function.

$$\begin{aligned} f(\alpha,\theta) &= 0 \text{ if any inhibitory inputs are active} \\ &= 0 \text{ if } \alpha < \theta \\ &= 1 \text{ if } \alpha \geq \theta \end{aligned}$$

The term "squashing function" is derived from the fact that the output function maps its input, $-\infty < \Sigma x_j < \infty$, into the output range, $0 < f(\Sigma x_j) < 1$.

If one or more of the inhibitory inputs are firing at time t, then the cell will not fire at time $t+1$, otherwise, the excitatory inputs are summed and compared to the cells activation threshold. Should the summed input activation, α, exceed the threshold, then the cell will produce an output signal at time $t+1$. This rather simple, nonlinear computation on the sum of all inputs can be used to generate a wide variety of useful computational elements on which a more general computing machine can be based. Figure 8.2 illustrates the use of cellular threshold and inhibitory inputs to produce *formal* neurons — that is, non-linear models of signal flow through a simple neuron that yield logical operators.

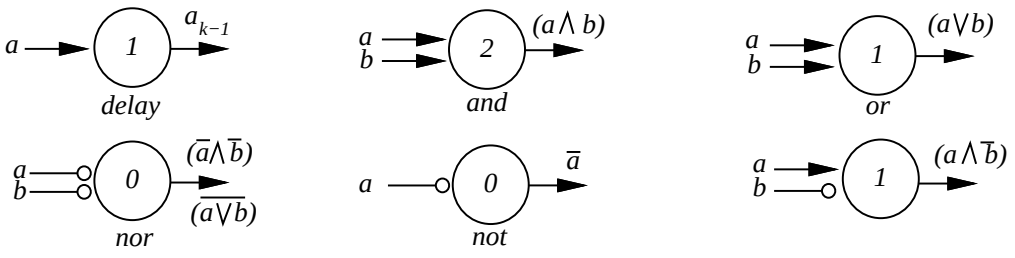


Figure 8.2 *McCulloch-Pitts Logical Operators*

The delay operator can be viewed as a sort of memory — it "remembers" its input signal for exactly one time step. A cascade of n such elements can remember a sequence of n input signals. Consider a 2 bit memory thus constructed and its equivalent finite state machine implementation. Figure 8.3 illustrates the two alternative "memories." The MP neuron implementation introduces a $2\Delta t$ latency from input S (Stimulus) to output R (Response) but the sequence of two states is obtained by simply tapping the signal at various places in the MP net. To remember the sequence of two binary inputs in an equivalent finite state implementation, we need a state for each of the signal sequences possible, or 2^n states. Therefore, the MP machine requires significantly fewer "units" to realize this form of memory. This economy is not uniform, but is a function of the application. Consider the problem of implementing a *binary scaler* (a divide by 2 network, see Figure 8.4).

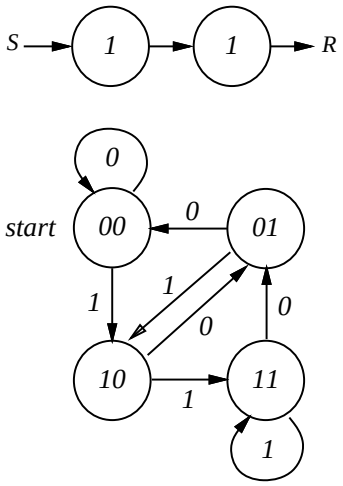


Figure 8.3 *The MP and FSM machine implementations of a 2 Bit Memory*

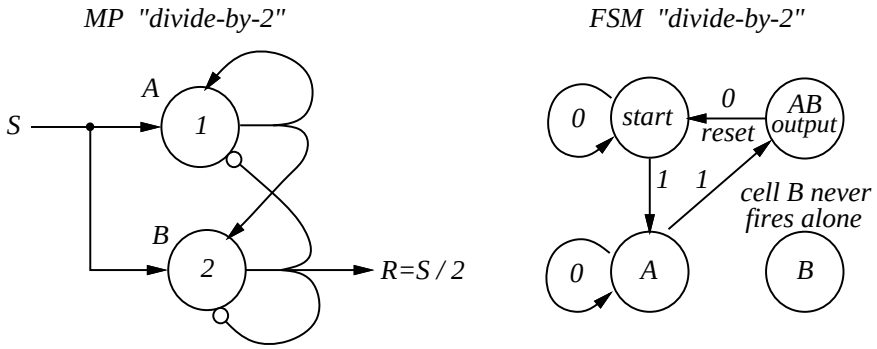


Figure 8.4 *The MP and FSM machine implementations of a Binary Scaler*

The first incoming pulse on the input line of the MP machine is ignored by cell B since it doesn't meet the threshold, but cell A produces an output pulse during the next time step. Notice that cell A is self-excitatory in that this output pulse is fed back to cell A. This excitatory feedback will *latch* cell A "on" until an inhibitory input can "reset" the cell. Moreover, cell A's output is also projected to cell B. Now another input pulse will add to cell A's output and cause cell B to fire — producing a pulse on the output line.

If we consider each unique state of cell A and cell B to have a corresponding state in the finite state machine implementation, we get the FSM illustrated in Figure 8.4 for the "divide-by-2" problem. From the start state, a pulse causes the transition to the "A" state. The next pulse that arrives causes the transition to state "AB".

Since cell B in the MP machine produces an inhibitory output, in effect resetting both itself and cell A, the next state following "AB" must be the "start" state regardless of the current input signal. The effect in either machine is to transform two input pulses into one output pulse and in so doing, to reset the machine. The state where both cell A and cell B are generating output is always followed by a "reset" (self inhibitory connection) and this reset takes a finite amount of time. If another pulse arrives on the input line during the reset computation, it will be lost — this is a bug in both the MP and FSM implementations (notice that the FSM implementation shows no transition on input "1" from state "AB").

To correct this problem, we note that another state in the FSM must be used to *catch* the discarded input pulses in these situations. Introduce a state "C" in the finite state machine as illustrated in Figure 8.5. The state catches the problematic pulse and introduces a transition on "1" from state "AB". It is protected in state "C" until state "A" is once again accessible. This state "C" requires another MP cell to be introduced as depicted in Figure 8.5. The output pulse generated by cell B now serves both to reset cells A and B and to activate cell C provided that one of our pathological pulses is present on the input. Notice also, that every state in the revised FSM has a transition on both "0" and "1".

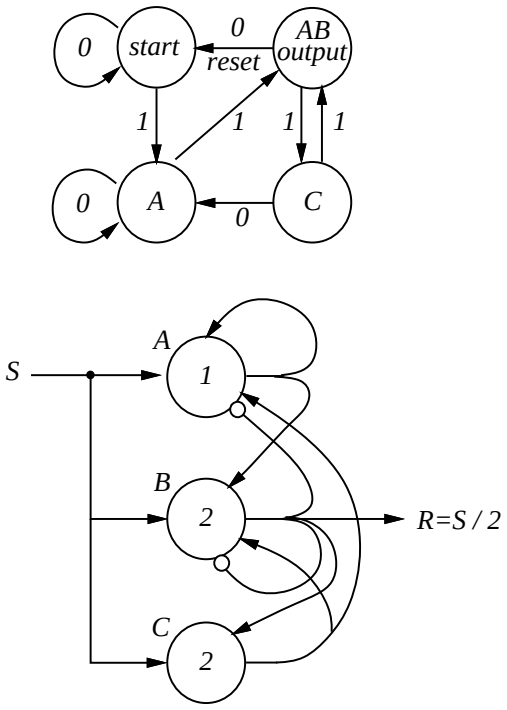


Figure 8.5 *A Corrected Binary Scaler*

8.1.1 The Canonical McCulloch-Pitts Machine

The foregoing discussion suggests that there is possibly a formal relationship between McCulloch-Pitts Machines and the Finite State Machine. In fact, such a relationship has been expressed in terms of the *canonical MP network*. This result is really quite significant, since it establishes an equivalent computability. Together with the results of Turing and Church, this result suggests that anything computable is in fact computable on a network of neuron-like elements such as those proposed by McCulloch and Pitts.

The problem that we address is that of constructing an MP network, N^M that is equivalent to the FSM, M. M is defined as follows:

M : m inputs, $(S_1, S_2, ..., S_m)$,
n outputs, $(R_1, R_2, ..., R_n)$,
p states, $(Q_1, Q_2, ..., Q_p)$,
$G(Q_i, S_j)$ state transition function, and
$F(Q_i, S_j)$ output function.

For the sake of the construction, we will define a state of the MP machine to be an element of the set of all possible cell firing patterns as before. The MP network equivalent of the finite state machine will have:

N^M : m input fibers, $(s_1, s_2, ..., s_m)$,
n output fibers, $(r_1, r_2, ..., r_n)$, and
$m \times p$ cells, m cells for each state, Q_j, of M.

The construction is defined by the following sequence of steps:

1. make an $m \times p$ cell array, these cells are "ANDS" with thresholds set to 2,
2. construct m input fibers, one for each row of the array,
3. for each of the p columns of cells, construct an m fiber input bus and m ascending and descending output fibers,
4. construct the state transition function of M, $G(Q_i, S_j)$, on the ascending output fibers of N^M, and
5. construct the output function of M, $F(Q_i, S_j)$, on the descending output fibers of N^M by projecting output fibers onto an "OR" cell.

EXAMPLE: Consider the FSM illustrated in Figure 8.6. This FSM produces an output when a 0 input signal follows a sequence of two or more 1 inputs. It could be signaling the trailing edge of a square wave input that has duration greater than a preset threshold, for example.

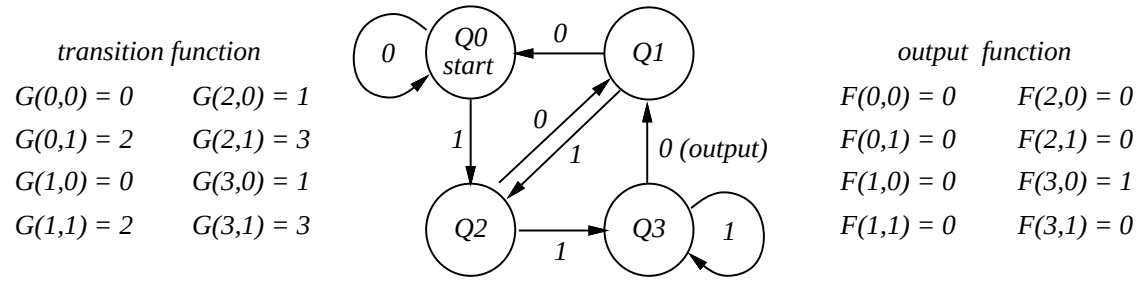
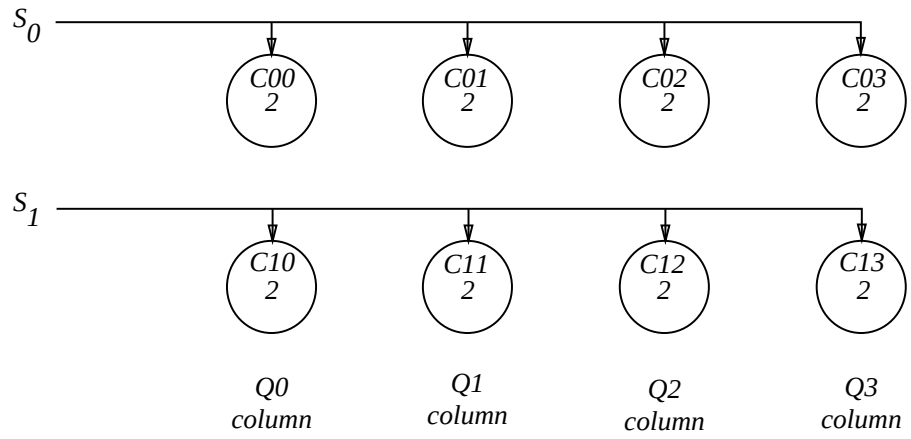


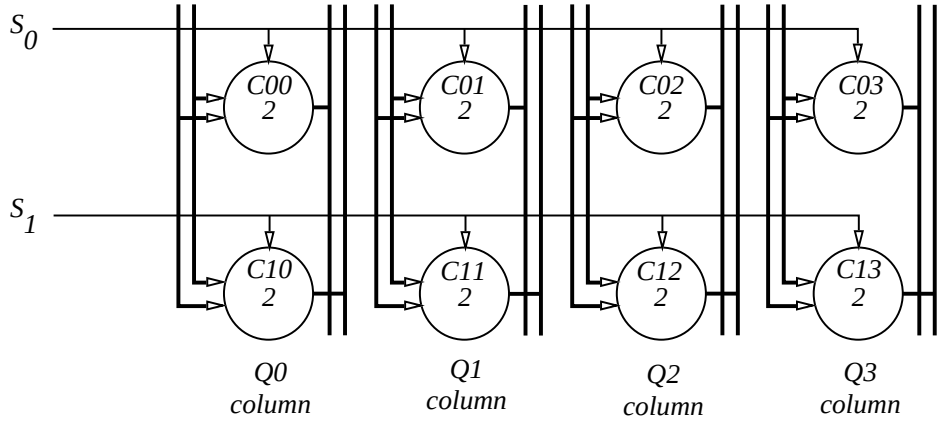
Figure 8.6 *The FSM for Trailing Edge Detection*

1. **make an $m \times p$ "AND"cell array,**
   This FSM has $m = 2$ possible input signals (0 and 1) and $p = 4$ states (so we must construct a 2×4 array of MP cells) and,

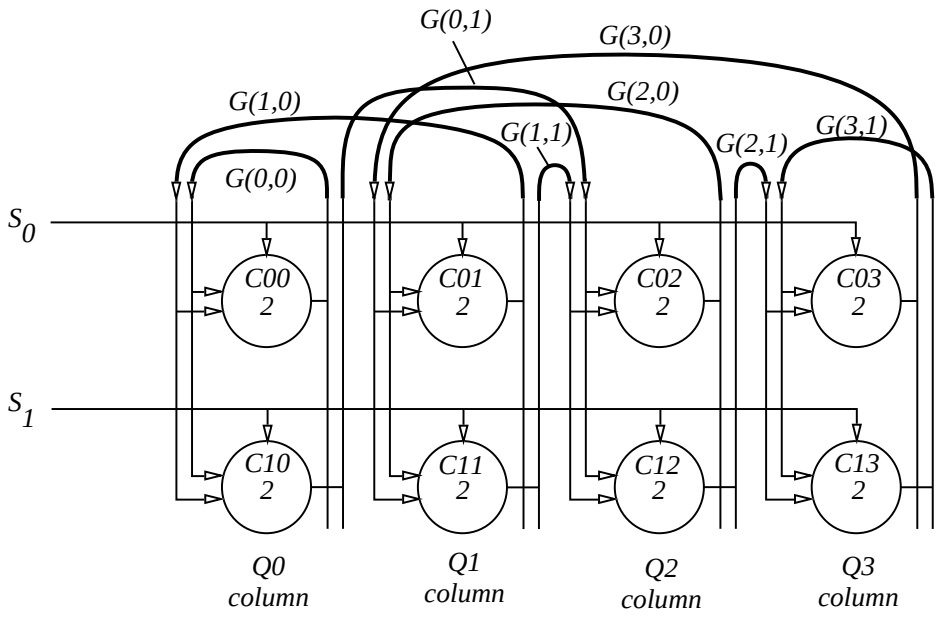
2. **construct m input fibers, one for each row or the array,**



3. **for each of the p columns of cells, construct an m fiber input bus and m ascending and descending output fibers,**



4. **construct the state transition function of M, $\mathbf{G}(Q_i, S_j)$, on the ascending output fibers of N^M,**


G(0,1)
G(3,0)
G(1,0)
G(2,0)
G(1,1)
G(2,1)
G(3,1)
G(0,0)
S0
S1
C00 2
C01 2
C02 2
C03 2
C10 2
C11 2
C12 2
C13 2
Q0 column
Q1 column
Q2 column
Q3 column

5. construct the output function of M, $\mathrm{F}(Q_i, S_j)$, on the descending output fibers of N^M.

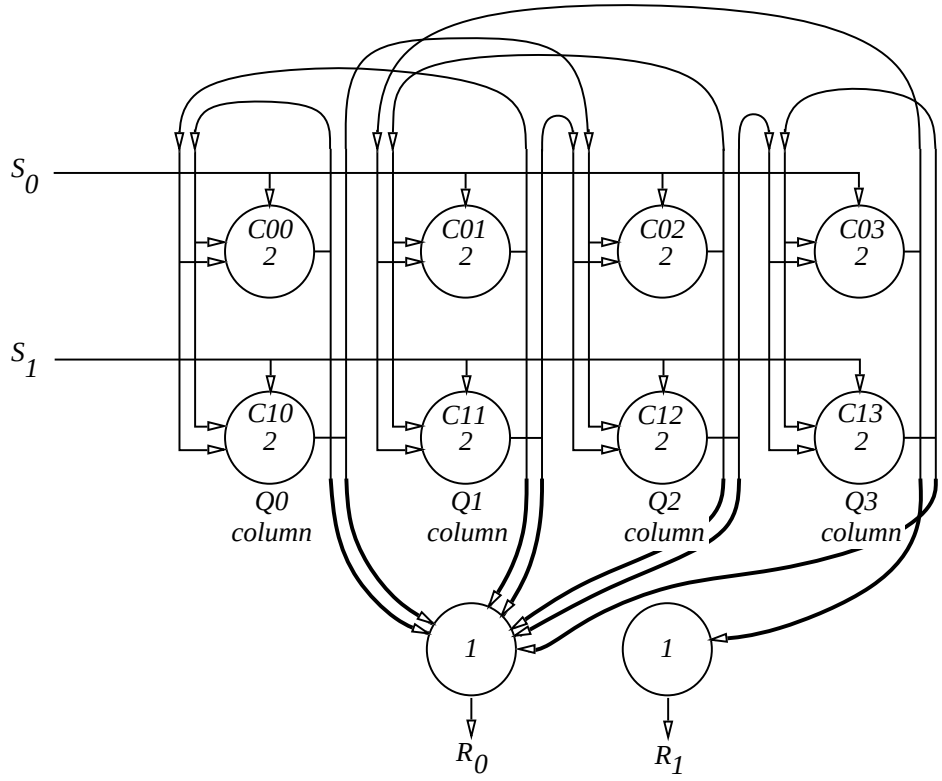


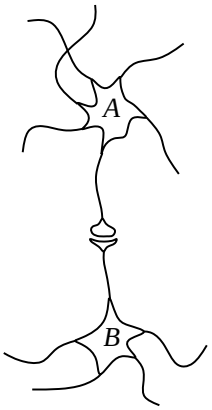
Figure 8.7 *The Equivalent McCulloch-Pitts Network for the Trailing Edge Detector.*

8.2 The Correlational Synapse

Let's assume that learning can be viewed as the problem of ascertaining the underlying *causality* in the environment. Understanding cause and effect relationships in the world is fundamental to choosing actions in response to sensations. The idea that outcome can be correlated to action has been noted by Aristotle, who asserted that two events are correlated if "...one event is of a nature to occur after another."

Most neuroscientists think that memory is due to *neuronal plasticity* which refers to the manner in which neurons can change structurally or functionally. This form of adaptation can produce changes which last for a lifetime, or which are forgotten or overwritten in rather short periods of time.

A largely accepted hypothesis postulates that *information is stored in synaptic changes and that these produce changes in the way in which each cell communicates with other cells in the brain.* Hebb realized that this basic objective in an adaptable system could be used to model the process of adaptation in the brain:



> "...when an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency as one of the cells firing B, is increased."

This observation models learning as a cause and effect correlation between a pre- and a post-synaptic cell. It is generally understood that changes in the synaptic resistance and capacitance encode the correlation between cells A and B.

8.2.1 Associative Memory

The associative memory task involves storing pairs of signal patterns (representing *events*) in such a way as to cause the first signal pattern to elicit the second signal pattern. This kind of memory is said to be *content addressable* if presentation of a fragment of the first pattern (or a corrupted/noisy version) elicits the whole of the second pattern. A special case of content addressable memory is the *autoassociative* memory, in which the first and second signal patterns are the same. In this case, the problem is to recall a complete pattern from an incomplete version of it — a problem sometimes referred to as pattern completion.

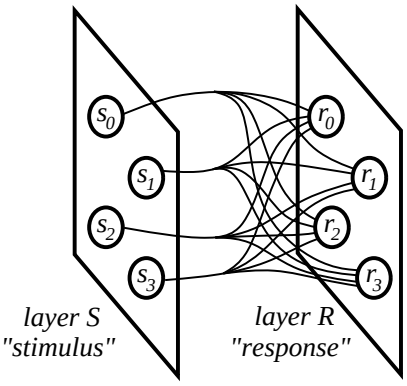


Figure 8.8 *A Correlation Matrix Memory*

Consider two layers of neurons connected in a network as shown in Figure 8.8. Each of n cells in input layer S connects to all n cells in output layer R. The synaptic *connectivity* matrix, C, describes the synaptic strength for each connection. $C(i,j)$ defines the synapse between cell j in layer S and cell i in layer R. These synaptic weights can be defined using a simple learning rule based on the Hebb synapse. If $\bar{s}$ is an n dimensional, **unit length** input column vector, and $\vec{r}$ is the corresponding n dimensional output column vector, then the connectivity matrix necessary to relate $\vec{r}$ to $\bar{s}$ is defined by:

$$\Delta C_k = \eta \vec{r}_k \bar{s}_k^T \tag{8.1}$$

where η is a learning rate parameter. The vector product $\vec{r}\bar{s}^T$ is called the *outer product.* The particular input/output pair, $(\bar{s}, \vec{r})_k$ used to generate ΔC_k is referred to as a training instance and in general, a single correlation matrix can be used to represent many (consistent) training instances.

$$\begin{array}{ccc} (\bar{s}_1, \vec{r}_1), & \ldots & (\bar{s}_k, \vec{r}_k) \\ \Downarrow & & \Downarrow \\ \Delta C_1 = \vec{r}_1 \bar{s}_1^T & \ldots & \Delta C_k = \vec{r}_k \bar{s}_k^T \end{array}$$

The resulting correlational memory recalls the output patterns associated with each input pattern through the relation,

$$\vec{r}_{n\times 1} = C_{n\times n} \bar{s}_{n\times 1} \tag{8.2}$$

where,

$$C = \sum_i \Delta C_i \tag{8.3}$$

Now, any memory key in the training set can be used to elicit the correct output response *using the same associative memory:*

$$\vec{r} = C\bar{s}. \tag{8.4}$$

It is important in this model that the set of all input vectors (memory keys) form an *orthonormal basis* for the space of all memory keys. They must form a basis because the associative memory cannot produce recollections for which it has no training instance. They must be orthogonal since multiple exposures to non-orthogonal training inputs will *add* continually into the associative memory (Equation 8.3) creating output recollections with excessively large magnitudes. And, all input vectors must be normalized, since the training and recollection cycle (Equation 8.1 and Equation 8.4) would introduce a scaling factor equal to the square of the magnitude of $\vec{s}$, otherwise.

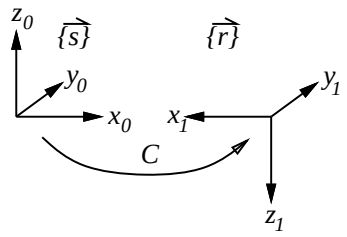
This model is extremely simple, but it possesses several very important properties. It is clear that every synapse in the connectivity matrix contributes to input/output correlation for several training instances. Therefore,

1. the information in the connectivity matrix, S, is not localizable,
2. the operation of many synapses is required to perform the mapping from input to output, and
3. information is distributed across many synaptic strengths, so that some fraction of these synapses can be removed without catastrophic failures in the memory — it tends to degrade gracefully.

These characteristics are sometimes referred to as *holographic* properties of an associative memory.

EXAMPLE: An important problem in vision and robotics involves mapping one coordinate system into another. We will see many instances of this problem in later chapters. In this example,

we will demonstrate that this problem can be *learned* incrementally from a sequence of training pairs as described above. Consider the coordinate transformation illustrated below.



In this example, coordinate frame 1 is displaced by a pure rotation from frame 0 (the diagram also indicates a translation for clarity, ignore it). The set of input vectors, $\{\bar{s}\} = \{\bar{x}_0, \bar{y}_0, \bar{z}_0\}$, consists of the basis vectors for coordinate frame 0 and the output vectors, $\{\bar{r}\} = \{\bar{x}_1, \bar{y}_1, \bar{z}_1\}$, consists of the basis vectors for coordinate frame 1. We wish to learn the transformation matrix, C that maps x_0 to x_1, y_0 to y_1, and z_0 to z_1. We will set the learning rate parameter $\eta = 1$, in this example, so that the transform is learned in one exposure to the training set.

Three mappings must be represented simultaneously in the associative memory, the first of which is the transformation of the $\bar{x}$ axis. For this case,

$$\Delta C_x = \eta \bar{r}_x \bar{s}_x^T = \eta \bar{x}_1 \bar{x}_0^T = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} [1\ 0\ 0] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{8.5}$$

Similarly,

$$\Delta C_y = \eta \bar{r}_y \bar{s}_y^T = \eta \bar{y}_1 \bar{y}_0^T = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} [0\ 1\ 0] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{8.6}$$

and,

$$\Delta C_z = \eta \bar{r}_z \bar{s}_z^T = \eta \bar{z}_1 \bar{z}_0^T = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} [0\ 0\ 1] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{8.7}$$

Now,

$$C = \sum_{i=x,y,z} C_i = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{8.8}$$

This transform expresses a linear transformation from coordinate frame 0 to coordinate frame 1 using 3 training pairs. Figure 8.9 illustrates the network that executes this transformation.

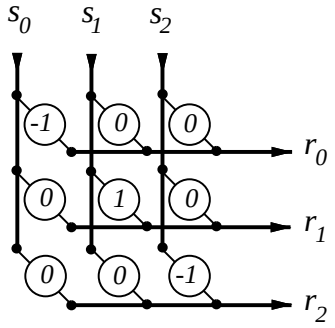


Figure 8.9 *The Solution: A Linear Associative Transformation*

We should, therefore, be able to recollect the correct output vectors from the input training vectors.

$$\begin{matrix} \bar{r}_x & \stackrel{?}{=} & C\bar{s}_x \end{matrix}$$

$$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \surd$$

$$\bar{r}_y \stackrel{?}{=} C\bar{s}_y$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \surd$$

$$\bar{r}_x \stackrel{?}{=} C\bar{s}_x$$

$$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \surd$$

Moreover, these training pairs form an orthonormal basis for Cartesian three space so we would expect the same transform to correctly transform any vector expressed in frame 0 to its equivalent representation in frame 1.

$$\begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \surd \tag{8.9}$$

This is a very simple demonstration of the ability of this linear associative memory to *generalize* to inputs it has not previously encountered.

8.2.2 Hamming Network

Suppose that we are confronted with the pattern on input illustrated in Figure 8.10. This is consistent with the aforementioned notion of an input pattern if you unwind the two dimensional signal into a longer one dimensional signal (these 7×7 arrays become vectors of length 49).

We will designate a pattern class or *exemplar*, s to be an N dimensional vector, and s_i^j designates the i^{th} bit of the j^{th} exemplar pattern. The associative memory problem will be to identify the exemplar s^j, $j = 1, M$ which is most closely associated with an input pattern, x. If Figure 8.10 represents two exemplar patterns, then the associative memory must recall the pattern from this set which most closely matches an input pattern. The classification procedure should use global evidence to overcome local defects (noise) or to complete the pattern when given a fragment of an exemplar vector The optimum minimum error classifier involves computing the *Hamming distance* from the input pattern to the exemplar for each class and then selects that class which is "closest." The Hamming distance is simply the number of bits in the input pattern which do not match the corresponding bits in the exemplar. A simple feedforward network can be implemented which computes the Hamming distance and then decides which interpretation yields the best match. The computational element we will use is slightly different than the classical MP neuron. In this model, inhibitory inputs are treated in the same way as excitatory inputs — that is, an inhibitory input is a negative input that decreases the net input activation level rather than shutting it off completely.

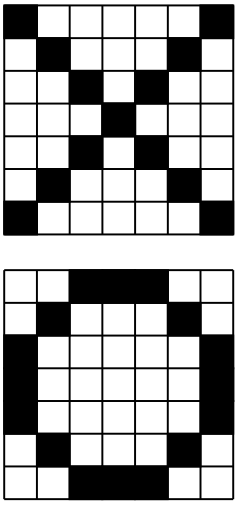


Figure 8.10 *Two Exemplar Patterns*

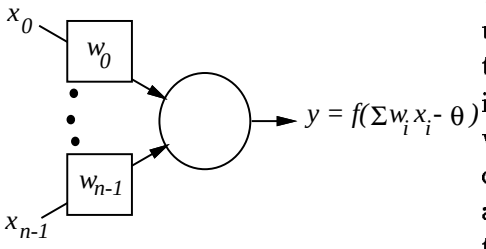


Figure 8.11 *The Artificial Neuron*

The resulting computational element is illustrated in Figure 8.11. The sum of the weighted inputs is processed by the squashing function to produce the output of the cell; θ is the threshold of the node, and w_i is the weight associated with the i^{th} input signal. The function f is the local processing executed by the node. Several candidate functions are illustrated in Figure 8.12. The Hamming net will use the threshold logic function to produce a graded output proportional to the Hamming distance between an input and an exemplar.

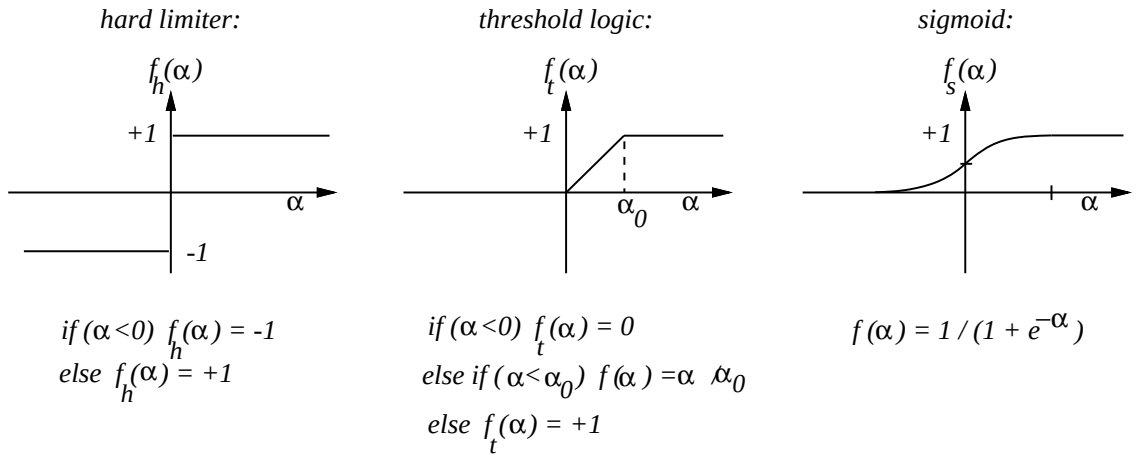


Figure 8.12 *Various Local Non-Linearities Employed by Artificial Neurons*

The Hamming network consists of M of the units depicted in Figure 8.11, one for each of the unique exemplars recognized by the network. Each of N bits of an input pattern is connected to each of these M units. Figure 8.13 illustrates this network structure.

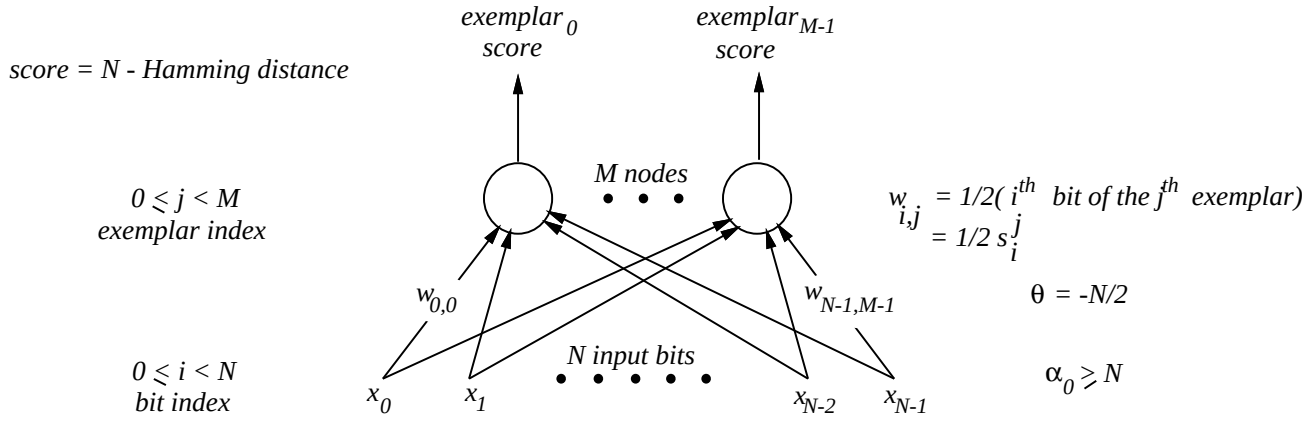


Figure 8.13 *A Network Which Computes the Hamming Distance*

This choice of weights and thresholds in the net computes $N - Hamming\ distance$ for the input pattern and each exemplar recognized by the system. The input weights from the input pattern to a node representing exemplar k, for example, effectively compute the *inner product* of the input

pattern and exemplar k. The factor of $1/2$ in the weight definition guarantees that:

$$-\frac{N}{2} \leq \sum_i w_{ik} x_i \leq \frac{N}{2}$$

for the k^{th} exemplar. Considering the threshold at node k,

$$0 \leq \sum_i w_{ik} x_i - \theta_k \leq N$$

producing as a score for exemplar k,

$$0 \leq f(\sum_i w_{ik} x_i - \theta_k, \alpha_0) \leq 1.$$

Therefore, a perfect match produces a score of 1 when $\alpha_0 = N$ and an input pattern which is the complement of an exemplar pattern produces a score of 0.

The MaxNet: Once the distance from an input pattern to each of the exemplars is computed, it remains to determine which of the resulting scores is the best match. One solution to this problem is to use a comparator on the output of the Hamming network. Such devices can be built from artificial neurons (or MP machines), but in this section we will discuss an iterative device which employs self-excitation and lateral inhibition to generate a "winner-take-all" network[1]. The network structure used in this network is illustrated in Figure 8.14.

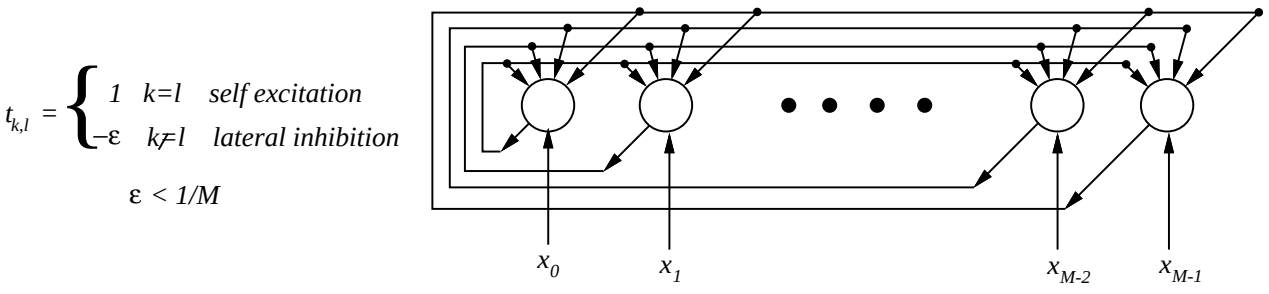


Figure 8.14 *An Iterative Implementation of the "Winner-Take-All" Network*

The inputs x_i initialize the network at time 0 and are subsequently recomputed in the net at each time step. The weights, $t_{k,l}$ connect the output of node k to an input of node l so that k's

[1]Part of the reason that we present this model, rather than others, is that the same network architecture can be used to implement the Hopfield network in Section 8.2.3.

output at time T contributes positively to its output at time $T+1$ ($k = l \Rightarrow t_{k,l} = 1$), while all other nodes tend to erode the activation of node k by an amount proportional to their activation ($k \neq l \Rightarrow t_{k,l} = -\epsilon$). The process continues iteratively until only one positive activation is left — the winner.

8.2.3 Hopfield Network

In 1982, Hopfield wrote a paper which viewed the pattern classification problem as a dynamic relaxation from an input pattern into one of the exemplars. The concept treats the exemplar patterns as *attractors* in the set of all possible patterns (sometimes referred to as the *configuration space* of the pattern). Unlike the Hamming network, this approach uses a McCulloch-Pitts network to solve the problem. Given an input pattern, x^k, at time k, this approach maps the pattern to x^{k+1} through a set of MP weights, $[t_{i,j}]$.

An idealized representation of this problem [adapted from Hertz, Krough, and Palmer] in Figure 8.15 shows schematically that the hopfield network relies on the dynamics of the classification procedure to modify a pattern continuously from any pattern configuration toward an exemplar. This model will use values of $+1$ for black bits (firing neurons) and -1 for white bits (not firing) in the pattern and will use the *hard limiter* squashing function illustrated in Figure 8.12. The operation of the Hopfield network is dynamic which means that when an input pattern, x, within a particular basin of attraction is applied, the network will perform a sequence of iterative modifications to the pattern which transform the input pattern continously into the convergent pattern for this basin of attraction.

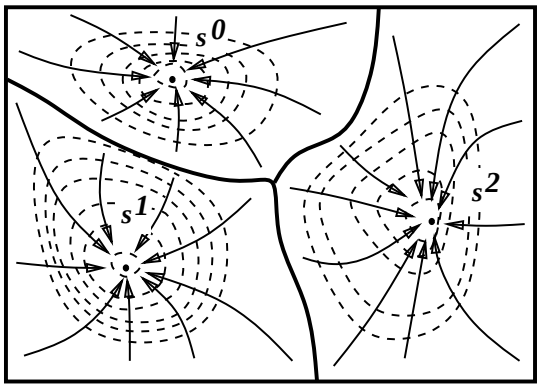


Figure 8.15 *Basins of Attraction Within the Pattern Configuration Space*

The Hopfield network (illustrated in Figure 8.16) is the same structure as the MaxNet portion of the Hamming network described earlier. Notice that the weights are defined to reflect *intra*pattern correlations. This is reminiscent of the correlational synapse model since if bit i and bit j are similarly correlated over many exemplars, then the network will express this as a relatively large synaptic weight, $t_{i,j}$. To operate the network, the output of each cell is initialized with the input pattern, x^0. The pattern is then modified iteratively until convergence,

$$x_i^{k+1} = f_h\left(\sum_{j=0}^{N-1} t_{i,j} x_j^k\right),\ 0 \leq j \leq N-1.$$

$$t_{i,j} = 1/N \sum_{k}^{M-1} s_i^k s_j^k \qquad i \neq j$$

$$t_{i,j} = 1 \qquad i = j$$

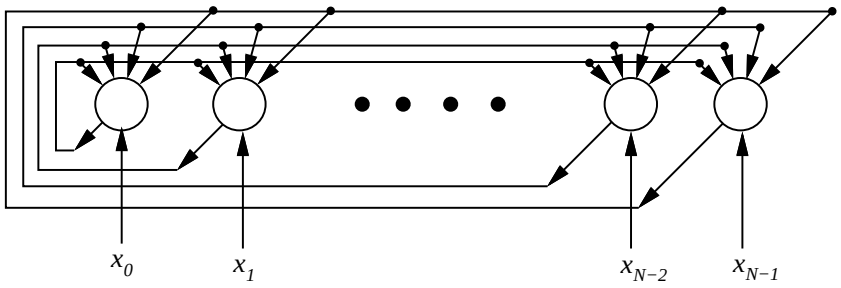


Figure 8.16 *The Hopfield Network for Pattern Classification*

Two issues are critical when analyzing the performance of the Hopfield network as an associative memory: **convergence** and exemplar **stability**. Convergence implies that an input pattern must eventually stop changing and generate one of a set of possible equilibria. An exemplar is a stable equilibrium when it is not altered by the recursive application of the Hopfield net. We will begin the discussion by demonstrating that the Hopfield network must converge, and then discuss the relationship between the equilibria and the exemplars.

Convergence

Consider the quadratic **energy** function expressing the *strain* between the Hopfield correlation matrix and the current pattern,

$$J = -\sum_{i,j} t_{i,j} x_i x_j .$$

It is clear that a perfect match between the pattern x and the correlation matrix t yields the minimal J and that any bitwise mismatches increase the energy of the pattern. Now, imagine that we know ahead of time that the correct value for x_i is x_i^*. Then the difference between the minimal J^* and the current J is

$$J^* - J = -\sum_{j \neq i} t_{i,j} x_i^* x_j + \sum_{j \neq i} t_{i,j} x_i x_j . \tag{8.10}$$

If $x_i^* = x_i$, the slope of energy with respect to x_i is zero, however, if $x_i^* = -x_i$, then

$$\begin{aligned} J^* - J &= 2x_i \sum_{j \neq i} t_{i,j} x_j \\ &= 2x_i \sum_{j} t_{i,j} x_j - 2t_{i,i} \end{aligned} \tag{8.11}$$

Therefore, under the assumption that x_i should be flipped, both the first and second terms of Equation 8.11 are negative definite. The energy of the mismatch will be decreased or left unchanged by the action of the Hopfield net until the pattern reaches a least-squares optimal match to the correlation matrix.

This property of the algorithm (sometimes referred to as a *voting* algorithm) allows the network to reject small errors in the input pattern by driving the pattern toward the attractor. Note, however,

that there are actually two attractors *even when there is only one exemplar*. The other attractor is the pattern which is the complement of the exemplar pattern — the bitwise correlations used to define the weights are identical in this case to the weights derived from the original pattern. From an initial pattern, if more than half of the bits are different than the single exemplar pattern, the net will converge to this complement attractor rather than completing the exemplar pattern.

Stability

While the Hopfield net is guaranteed to stop altering the pattern eventually, we have not yet demonstrated that the equilibria so generated will bear any resemblance to the exemplar patterns. It can be demonstrated that the choice for the synaptic weights defined in Figure 8.16 produces a stable classifier for a single exemplar, by noting that when the exemplar pattern is presented as input

$$f_h(\sum_i t_{i,j} x_i) = x_j \ \ for\ all\ j.$$

That is, the pattern does not diverge from the correct classification — if the exemplar pattern is put into the network, then the same pattern will come out.

This is clear if we examine the stability of a particular pattern, s^α. If this pattern is stable, then

$$f_h(\sum_i t_{i,j} s_i^\alpha) = s_j^\alpha \ \ for\ all\ j.$$

Recalling that $t_{i,j}$ is derived from the bitwise correlations over all exemplars, we get

$$f_h(\sum_i (\frac{1}{N} \sum_m s_i^m s_j^m) s_i^\alpha) = s_j^\alpha \ \ for\ all\ j.$$

Now, we can separate the sum into the terms for s^α and all the rest,

$$f_h(\frac{1}{N} s_i^\alpha s_j^\alpha s_i^\alpha + \frac{1}{N} \sum_i (\sum_m s_i^m s_j^m) s_i^\alpha) = s_j^\alpha \ \ for\ all\ j,$$

or, since $s_i^\alpha s_i^\alpha = +1$,

$$f_h(\frac{1}{N} \left[s_j^\alpha + \sum_i \sum_m s_i^m s_j^m s_i^\alpha \right]) = s_j^\alpha \ \ for\ all\ j. \tag{8.12}$$

If there are no other exemplars, $m = \alpha$, then Equation 8.12 becomes $f_h(\frac{2}{N} s_j^\alpha$, which is clearly equivalent to s_j^α for all j and the stability criterion is satisfied. However, the network will only be stable in general only if the second term is *small enough* not to change the sign of the argument to the hard limiter.

The second term of Equation 8.12 expresses the crosstalk between exemplar patterns in the content addressable memory. If M, the number of exemplar patterns, is small enough, then there is a good probability that all the exemplar patterns will be stable and will be surrounded by a

basin of attraction. It is intuitively obvious that if we continually add more exemplar patterns, eventually the network will saturate and cease to express the bitwise correlations for all exemplar patterns. Hopfield proposed a heuristic based on the probability that random exemplar patterns would interfere with one another which states that the content addressable memory will couple through the crosstalk terms when $M/N \geq 0.15$ — that is, the memory will saturate if the ratio of the number of exemplar patterns to the number of input bits (and therefore, the number of cells) exceeds 0.15.

8.3 The Perceptron Convergence Rule (PCR)

Let's return to the model of a neuron proposed in Figure 8.11 which computes the hard limiter function over the weighted sum of its inputs,

$$y = f_h \left[\left(\sum w_i x_i \right) - \theta \right].$$

This nonlinearity produces a decision which divides the input space into two distinct regions. Region "A" is that portion of the input space which produces a $+1$ activation for the cell, and region "B" is that region where the squashing function yields a -1 activation. If we have a cell with N input fibers, the result is an $N-1$ dimensional decision surface which distinguishes two, linearly separable regions of the N dimensional input space. This device is termed the *perceptron* and was first discussed by Rosenblatt in 1957.

Consider the two input perceptron illustrated in Figure 8.17. In this model, the decision boundary can be identified by the locus of points in the input space where the activation, α, is equal to zero. This is the boundary in input space where the output switches from region "A" to region "B." The equation of this boundary is defined by setting the activation of the cell to zero,

$$w_0 x_0 + w_1 x_1 - \theta = 0.$$

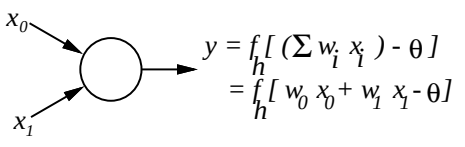


Figure 8.17 *A Two Input Linear Perceptron*

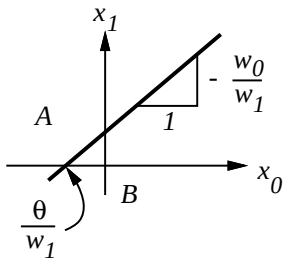


Figure 8.18 *The Linear Decision Boundary*

Solving this relation for x_1 as a function of x_0, w_0, $and w_1$ yields

$$x_1 = \frac{\theta}{w_1} - \frac{w_0}{w_1} x_0,$$

which is the standard slope/intercept equation of the line that divides the input space spanned by (x_0, x_1) into two half-spaces. The slope of this decision surface is $-\frac{w_0}{w_1}$ and its intercept is $\frac{\theta}{w_1}$. This parametric decision surface is the basis for a learning rule which can incrementally alter the cell parameters, w_0, w_1, *and* θ, so the output of the cell correctly classifies a set of input/output training instances.

A major shortcoming of the Hebbian learning rule (Equation 8.1) is that it is susceptible to overtraining. That is, if the sequence of orthogonal training relations is presented to the cell twice, the synaptic weights will be corrupted. This result is not intuitively pleasing. Clearly, the mapping from input to output should not degrade with additional experience. The obvious shortcomings of this "one-pass" learning mechanism lead to the Perceptron Convergence Rule (PCR):

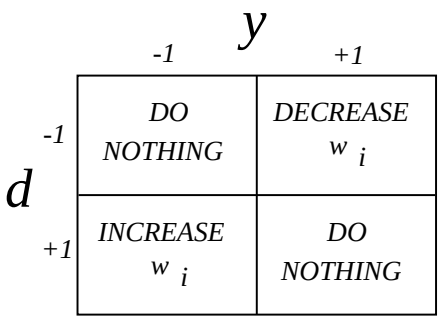
$$\begin{aligned} w_i(t+1) &= w_i(t) + \Delta w_i \\ \Delta w_i &= \eta x_i (d - y) \end{aligned} \tag{8.13}$$

where:

$$\begin{aligned} d &= \text{the desired output for input } \vec{x} \\ y &= \text{the current output.} \end{aligned}$$

This learning rule turns itself off when there is no longer any difference between the desired and actual outputs. Multiple exposures to the training set change the synaptic weights until learning is effectively turned off when the proper input/output relation is established. Because the learning rule is modulated by the difference between the desired and actual output, this rule is sometimes referred to as the *delta* rule.

The logic of this learning rule is clear when one examines the cases which cause a weight to be adjusted. Assuming that input bit $x_i = +1$, if the desired output is greater than the actual output, then the weight will be adjusted upward. Likewise, if the the desired output is smaller than the actual output, the weight will be adjusted downward. If the desired output matches the actual output, then none of the synaptic weights will be changed. This observation leads to the perceptron convergence theorem: if two classes are linearly separable, and if training instances are presented enough times, then eventually the weights will adapt to a sufficient decision boundary.



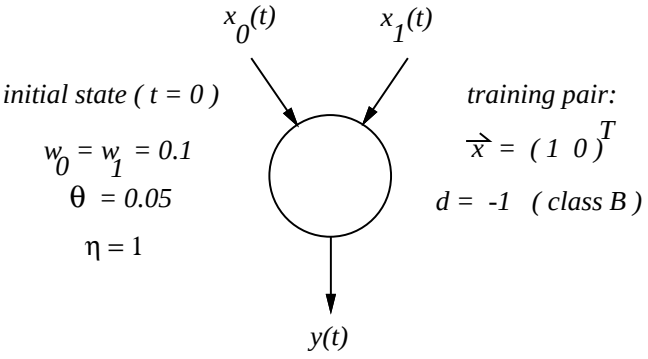
The algorithm for training the perceptron is then:

initialize weights ($w_i(0)$) and thresholds (θ_i) to small, random, non-zero values
until convergence {
  for each training instance $(\vec{x}_j, d_j)$ {
    evaluate $y(t) = f_h\left[\left(\sum_{i=0}^{n-1} w_i(t)x_{ji}\right) - \theta\right]$
    for all $i \in (0, n-1)$ $w_i(t+1) = w_i(t) + \eta\,[d_j - y(t)]\,x_{ji}$
  }
}

EXAMPLE: Consider the two input perceptron illustrated.

At time 0, this cell produces an incorrect output;

$$\begin{aligned} y(0) &= f_h\,[(0.1)(1) + (0.1)(0) - 0.05] \\ &= f_h\,[0.05] \\ &= +1. \end{aligned}$$



Since the actual output does not match the desired output, the synaptic weights are adjusted:

$$\begin{aligned} w_0(t+1) &= w_0(t) + \eta\,[d - y(t)]\,x_0(t) \\ &= 0.1 + 1(-1-1)1 = -1.9 \\ w_1(t+1) &= w_1(t) + \eta\,[d - y(t)]\,x_1(t) \\ &= 0.1 + 1(-1-1)0 = 0.1 \end{aligned}$$

so that, now

$$y(1) = f_h\,[(-1.9)(1) + (0.1)(0) - 0.05] = -1.$$

Therefore, after one presentation of the training data, this decision boundary correctly classifies the training data and adaptation of the synaptic weights will cease.

8.4 Least Mean Squared (LMS) Error or Delta Rule Learning

The LMS or Delta Learning rule is almost identical to the perceptron convergence rule except for the fact that the nonlinear hardlimiter is removed. This yields a differentiable learning rule for which we may derive a convergence proof. The learning rule becomes:

$$\begin{aligned} w_i(t+1) &= w_i(t) + \Delta w_i \\ \Delta w_i &= \eta x_i(d - \alpha) \end{aligned}$$

where:

$$\alpha = (\sum_{i=0}^{n-1} w_i(t)x_i(t)) - \theta$$

is the current activation of the cell.

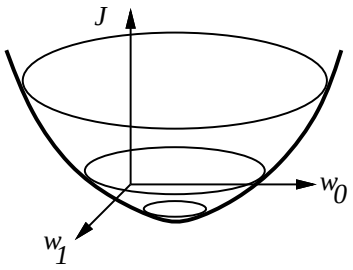


Figure 8.19 *The Quadratic Mean Squared Error Functional*

The mean squared error between the desired output and the current activation is used as the objective function, and we will derive a learning rule that minimizes this objective function with respect to the synaptic weights. The mean squared error between the desired and actual output is defined by Equation 8.14.

$$J(\vec{w}) = \frac{1}{K}\sum_{k=1}^{K}[d_k - \alpha_k]^2\,, \tag{8.14}$$

This function is illustrated schematically for two input weights in Figure 8.19. A gradient descent strategy for minimizing J follows the gradient $\partial J/\partial w$ downhill toward the minimum.

$$\Delta w_i = -c\frac{\partial J}{\partial w_i}$$

Differentiating Equation 8.14 yields:

$$\begin{aligned} \frac{\partial J}{\partial w_i} &= -\frac{2}{K}\sum_{k=1}^{K}(d_k - \alpha_k)\frac{\partial \alpha_k}{\partial w_i} \\ &= -\frac{2}{K}\sum_{k=1}^{K}(d_k - \alpha_k)x_{ki} \end{aligned}$$

so that the learning rule guaranteeing the minimum squared error is

$$\Delta w_i(t) = \frac{2c}{K} \sum_{k=1}^{K} (d_k - \alpha_k) x_{ki}(t). \tag{8.15}$$

This rule for adapting weights will eventually arrive at the global minimum of the objective function, but it requires all K training pairs. We may approximate this rule by considering only a single training pair:

$$\Delta w_i(t) \approx \eta (d_k - \alpha_k) x_{ki}(t). \tag{8.16}$$

8.5 Multilayered, Nonlinear Perceptrons: Backpropagation

A fundamental limitation of the perceptron is its inability to capture general decision boundaries: the perceptron cannot be trained to distinguish training sets which are not linearly separable. Minsky and Papert (reference) observed the inherent limitations of these adaptive units as a basis for computation by noting that they could not express the exclusive-or (XOR) operator — this logical operation requires more than a simple linear decision boundary.

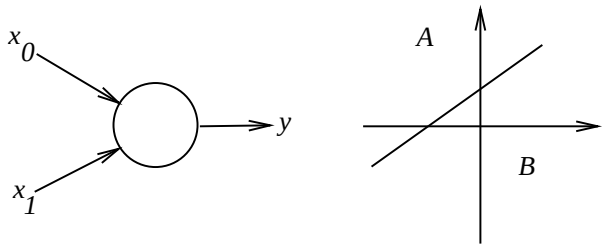


Figure 8.20 *The Single Layer Perceptron*

We already have the tools to address these limitations. Recall that in our discussion of MP neurons, we formulated MP units capable of behaving like AND and OR gates. Figure 8.21 illustrates how we might use the conjunction of linear decision boundaries to delimit a closed volume of the state space. This technique is capable of recognizing convex (possibly unbounded) regions in the state space. Concave regions cannot be expressed. In these situations, at least two of the linear decision boundaries must be inconsistent with one another.

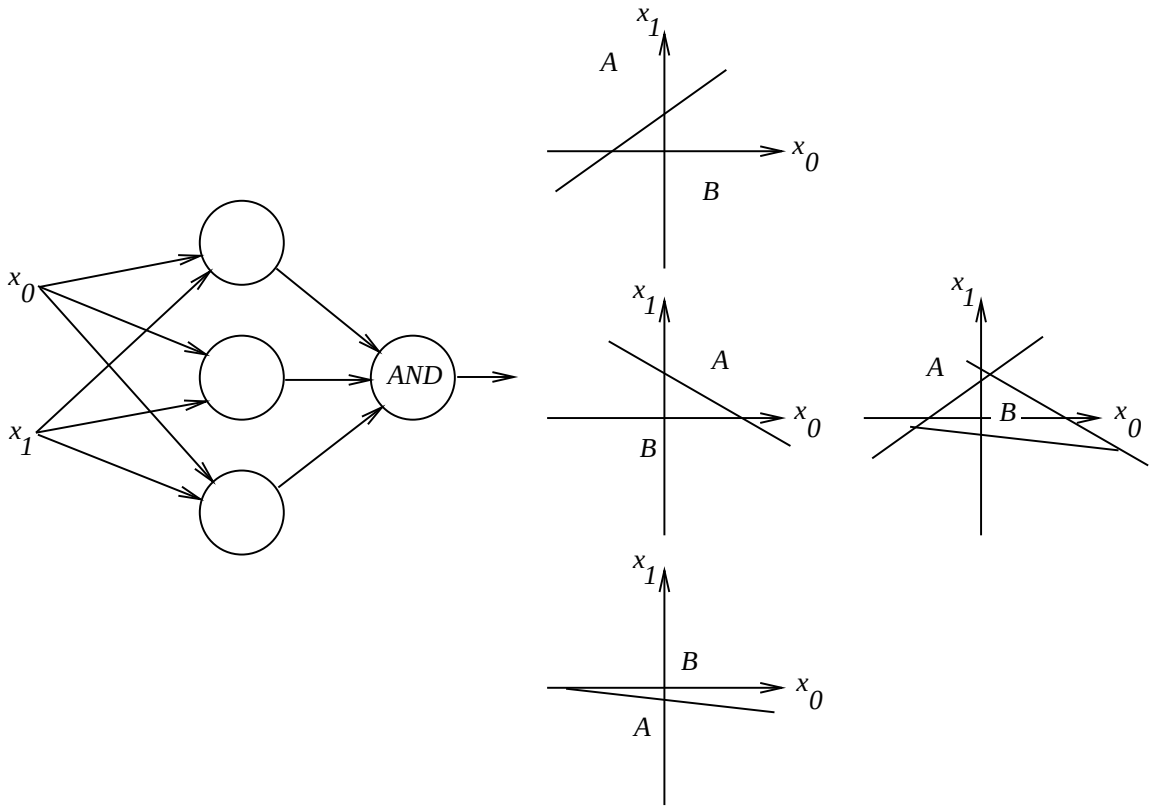


Figure 8.21 *The Two Layer Perceptron*

Figure 8.22 illustrates a machine constructed from a disjunction over a family of these two layer machines. This machine is capable of properly classifying any contiguous set of states in the state space.

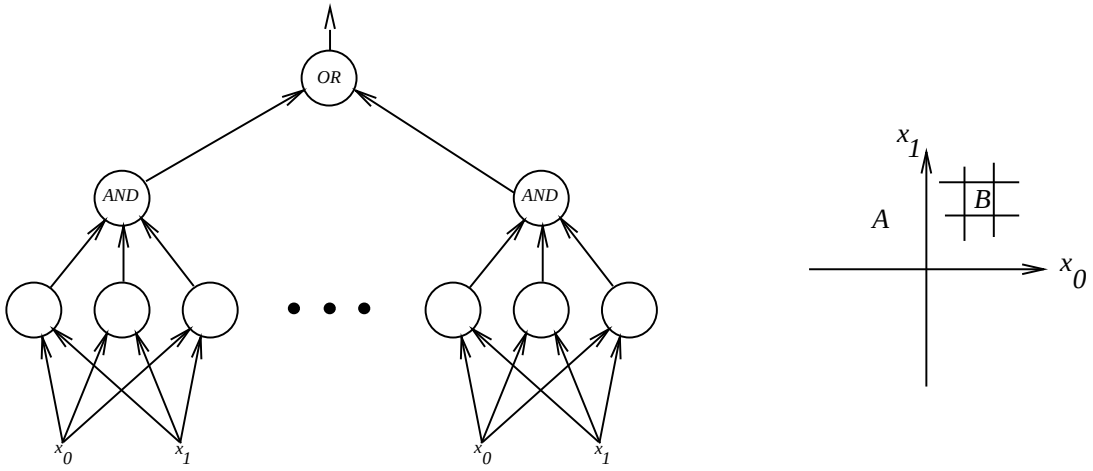


Figure 8.22 *The Three Layer Perceptron*

To see why, imagine a conjunction over four linear perceptrons that defines a small hypercubic volume in the state space. A disjunction over a family of these blobs of state space can be used to express any general region (concave, or non-contiguous) of the state space. This result is significant — three layers of units are sufficient to represent general decision boundaries!

However, having formulated the theoretical solution to expressing general decision surfaces, we are left with an even more serious problem. The Perceptron Convergence Rule (PCR) and the LMS learning rule are not designed to train more than one layer of perceptron units. These learning rules compare the actual output of the classifier with the desired output in order to modify the input weights to the unit. The LMS rule, however, provides the basis for generalizing this result to multiple layer machines. In this approach, we differentiated a quadratic error metric (involving a continuous squashing function) backward through the unit to determine the sensitivity of the output with respect to each of the input weights. In 1986, a technique for evaluating the sensitivity of the output of such networks to all three layers of input weights was formulated — multiple layer error backpropagation.

The canonical form of a multiple layered network is illustrated in Figure 8.23.The layer of units which are directly connected to input signals is refered to as the Input layer, the units connected to the output comprise the Output layer, and the intermediate units which are not directly connected to either input or output are referred to as the Hidden layer. The error metric we wish to minimize is the squared difference between the observed output per unit and the desired output

$$J = \frac{1}{2}\sum_{j \in O}(d_j - y_j)^2,$$

and the rule employed to adapt the weight from unit i to unit j will again simply descend the error gradient

$$\Delta w_{ij} = -c\frac{\partial J}{\partial w_{ij}}.$$

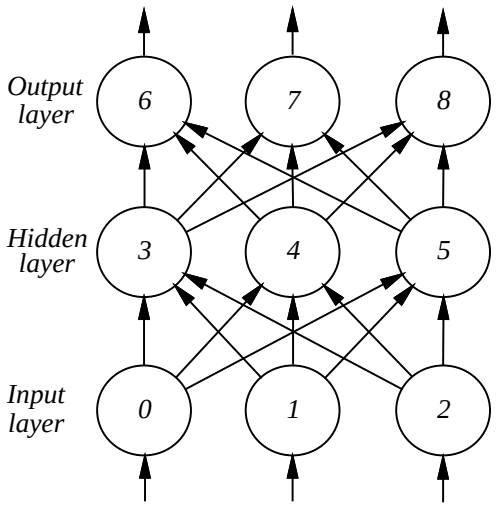


Figure 8.23 *The Canonical Multiple Layer Backpropagation Network.*

The gradient of the error metric with respect to input weights is computed using the chain rule[2]

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial \alpha_j}\frac{\partial \alpha_j}{\partial w_{ij}} \tag{8.17}$$

where α_j is the activation of unit j.

[2]Recall that given $x = f(y)$ and $y = g(z)$, $\frac{\partial x}{\partial z} = \frac{\partial x}{\partial y}\frac{\partial y}{\partial z}$.

The rightmost term in Equation 8.17 expresses how the activation of unit j varies with weight, w_{ij}

$$\frac{\partial}{\partial w_{ij}}[\alpha_j] = \frac{\partial}{\partial w_{ij}}\left[\sum_i w_{ij} y_i\right] = y_i. \tag{8.18}$$

The middle term on the right hand side of Equation 8.17 expresses the relationship between the output of unit j and the activation of unit j — this is just the squashing function. The squashing function employed is the sigmoid function

$$y = f(\alpha) = \frac{1}{(1+e^{-\alpha})}.$$

To differentiate this function with respect to the activation, α, let

$$\begin{array}{ll} x = 1 + e^{-\alpha} & \frac{\partial x}{\partial \alpha} = -e^{-\alpha} \\ y = x^{-1} & \frac{\partial y}{\partial x} = -x^{-2} \end{array}$$

then,

$$\begin{aligned} \frac{\partial y}{\partial \alpha} &= \frac{\partial y}{\partial x}\frac{\partial x}{\partial \alpha} = (-x^{-2})(-e^{-\alpha}) = \frac{e^{-\alpha}}{(1+e^{-\alpha})^2} \\ &= \frac{1}{(1+e^{-\alpha})}\frac{e^{-\alpha}}{(1+e^{-\alpha})} = f(\alpha)(1-f(\alpha)). \end{aligned} \tag{8.19}$$

To evaluate the first term on the righthand side of Equation 8.17, $\frac{\partial J}{\partial y_j}$, we must consider the case when unit j is an output unit and the case when unit j is a hidden unit.

OUTPUT UNITS:
($j \in$ *Output layer*)

$$\begin{aligned} \frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_{i\in O}(d_i - y_i)^2\right] = \sum_{i\in O}(d_i - y_i)\frac{\partial (d_i - y_i)}{\partial y_j} \\ &= -(d_j - y_j) \end{aligned} \tag{8.20}$$

HIDDEN UNITS: define O_j to be the output units affected by y_j, α_{O_j} to be the activations in the output layer affected by y_j ($\alpha_k, k \in O_j$). Then,
($j \in$ *Hidden layer*)

$$\begin{aligned} \frac{\partial J}{\partial y_j} &= \sum_{k\in O_j}\frac{\partial J}{\partial \alpha_k}\frac{\partial \alpha_k}{\partial y_j} = \sum_{k\in O_j}\frac{\partial J}{\partial \alpha_k}\frac{\partial}{\partial y_j}\left[\sum_i w_{ik} y_i\right] \\ &= \sum_{k\in O_j}\frac{\partial J}{\partial \alpha_k} w_{jk} \end{aligned} \tag{8.21}$$

Equations 8.17, 8.18, 8.19, 8.20, and 8.21 can be used to generalize the delta learning rule to networks with multiple layers. The update rule for the synaptic weight between units i and j can be written

$$\Delta w_{ij} = -c\delta_j y_i \tag{8.22}$$

$$\begin{array}{lrcl} \text{OUTPUT UNIT j:} & \delta_j & = & -(d_j - y_j)[f_s(\alpha_j)(1 - f_s(\alpha_j)] \\ \text{HIDDEN UNIT j:} & \delta_j & = & (\sum_{k \in O_j})[f_s(\alpha_j)(1 - f_s(\alpha_j)] \end{array} \tag{8.23}$$

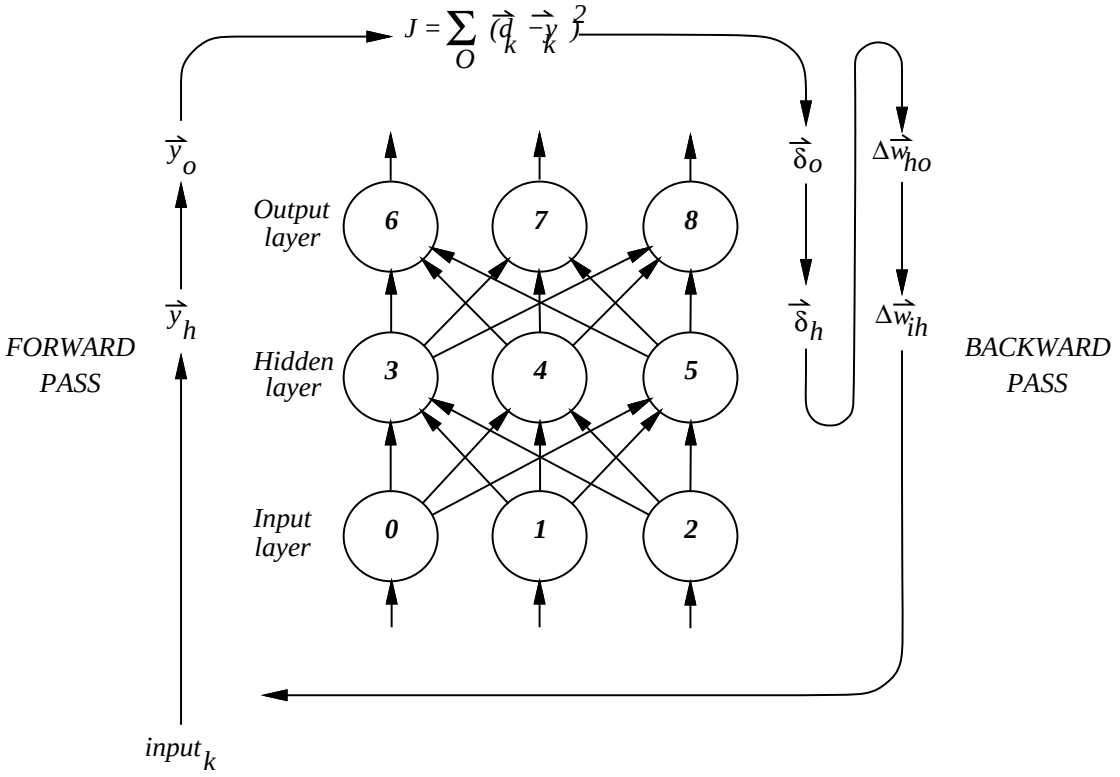


Figure 8.24 *A Schematic Algorithm for Error Backpropagation*

Figure 8.24 illustrates how this learning rule is used in the multiple layer network to propagate errors from the output layer backward throught the net. To apply this to a learning problem, this update cycle must be embedded within a supervisory program that manages the learning process. Typically, this involves the presentation of a set of training input/output relationships that we wish the network to learn. Because the learning process is incremental, many presentations of the training data are required. A set of n repetitions of these training data is referred to as an *epoch* in the learning process. After several epochs, the process will converge to the function from which the training data are derived. Pseudocode for training the multiple layer network using backpropagation can then be organized as follows:

```
procedure backprop()
{
```

```
    initialize(); /* define net configuration, initialize weights,*/
                  /* define learning rates, training data */
    while(!done && (epochs < MAX_EPOCHS)) {
        ++epochs;
        epoch(&mean_squared_error);
        if (mean_squared_error < ERROR_THRESHOLD) done = TRUE;
    }
}


procedure epoch(mean_squared_error)
{
    for (i=0; i<N_REPETITIONS; ++i)
        for (j=0; j<N_TRAINING_PAIRS; ++j)

            /* FORWARD PASS */
            for all units
                compute_output(unit)

            /* BACKWARD PASS */
            for all OUTPUT units
                mean_squared_error += (desired(j)-output(j))^2 / N_TRAINING_PAIRS
            from last OUTPUT unit to first HIDDEN unit
                compute_δ_j(unit)
            from last OUTPUT unit to first HIDDEN unit
                update_weights(unit)
}
```

References

Arbib, M. *, Perceptual structures and distributed motor control, Chapter 33: Handbook of Physiology — The system II.*

Hertz, J., Krogh, A., and Palmer, R. , *Introduction to the Theory of Neural Computation, Addison-Wesley Publishing Company, 1991.*

Lippman, R. , *An Introduction to Computing with Neural Nets, Artificial Neural Networks: Theoretical Concepts, The Computer Society Press, 1988.*

Minsky, M. , *Neural Networks: Automata made up of Parts Computation: Finite and Infinite Machines, Prentice Hall, 1967.*

8.6 Homework Exercises

1. **Canonical McCulloch-Pitts Network**
   Operate the McCulloch-Pitts network illustrated in Figure 8.7 and demonstrate that it computes the same function as the FSM in Figure 8.6. You must start the net from a well-defined starting state — with state *and* input specified. Start operating the network from state $C00$, that is assume that at time $t-1$ the output of cell $C00$ is high. From this initial state, apply the input 000110 to both the finite state machine and the McCulloch-Pitts equivalent. Record all the intermediate states explored during the operation of both machines.

2. **Correlational Synapse — Associative Memory**
   Consider this set of column vectors:

$$s_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad s_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \quad s_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \quad s_4 = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

   (a) Show that these vectors are mutually orthogonal, i.e., show that the dot product, $\vec{s_i} \cdot \vec{s_j} = 0$ for $i, j = 1, 2, 3, 4,\ i \neq j$.

   (b) Normalize each of the vectors by multiplying by a scalar, c, such that, $c\vec{s_i} \dot{c}\vec{s_i} = 1$ for $i = 1, 2, 3, 4$. The resulting vectors, $\bar{s}_i = c\vec{s_i}$, are orthonormal.

   (c) Now, suppose you want to form an associative memory in which vectors $\bar{s}$ are the *memory keys* and each $\bar{s}_i$ elicits the *recollection*, $\vec{r_i}$, where:

$$r_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad r_2 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \quad r_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad r_4 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

   Form the 3×4 matrix $A = \sum_{i=1}^{4} \vec{r_i}\bar{s}_i^T$, where $\bar{s}_i^T$ is the transpose of $\bar{s}_i$.

   (d) Show that the associative memory matrix, A, works.

3. **The Hopfield Net**
   This experiment examines the performance of the Hopfield Network. A graphics procedure is provided in the EdLab on the DECstation 5000's. You will find a makefile, several useful files, and the graphics procedures in the common directory.

   File "exemplars.c" contains several examples of input training patterns. The first two of the six classes provided are the exemplars for the "O" and the "X" classes, respectively. The next four patterns, patterns 3-6, are noisy versions of the X and the O, file "exemplars.c" provides a legend of the patterns.

   The code is written so that the network is trained over the first "NEXEMPLARS" of this pattern array. NEXEMPLARS is a compile time constant (whose value is initially set to 2) defined in file "net.h." This defines the training patterns for the network to be the exemplars for the O and X patterns. Try running the code as it is given to you. This requires that you execute the following commands:

   - make - *automatically recompiles and links modified files*
   - hopfield - executes the hopfield net program

   The program will proceed to train on the patterns designated by the NEXEMPLARS constant and will then ask you to input a selection from a user menu specifying which of the 6 given input patterns you wish to classify.

   As we will discuss in class, the performance of this network is expected to degrade as the ratio (number of patterns/number of nodes) approaches 0.15. With two training patterns and 7X7 patterns (49 nodes) this ratio is approximately 0.04. You should observe a relatively good performance when only two training patterns are used. You are to add exemplars. That is, design 5 more patterns you wish the network to classify, insert them as items 3 through 7 in the class array, and change NEXEMPLARS to 7. Note that procedure "apply_input_pattern()" will have to be changed accordingly. The new patterns should be as *orthogonal* as possible, that is, their inner products should be close to zero.

4. **The Hamming Net**
   Modify a copy of the Hopfield net code to function as a Hamming net. To do this, you should make your own Hamming directory and copy the Hopfield code into it. Rename and recode as required to implement the Hamming net in the graphics environment. We will discuss the Hamming net and the programming required at length in class. The Hamming net will also be tested on the 7X7 character recognition problem. Add exemplars and repeat the experiments conducted on the Hopfield net.

5. **Backpropagation: XOR**
   The structure of a network for the XOR net is shown below.

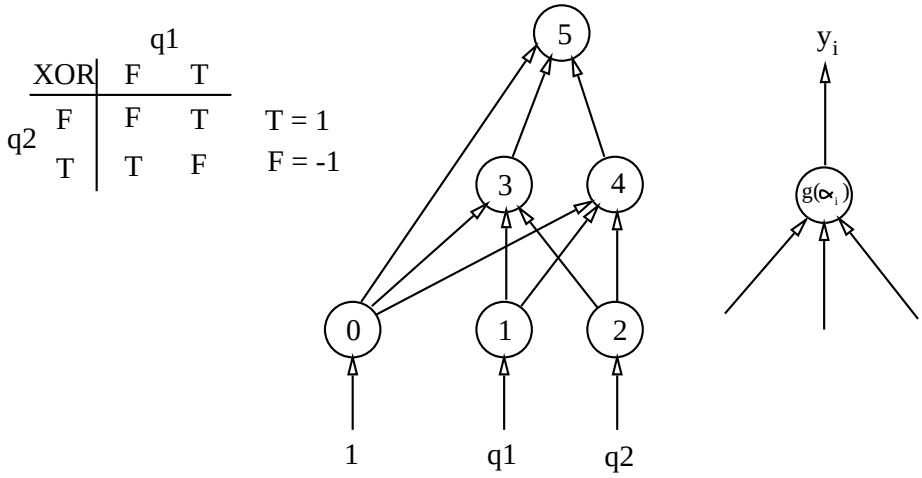


Figure 8.25 *The network structure for the XOR application*

Unit 0 is an input unit taking value 1 at all times and connected to units 3, 4, and 5. The weights on the connection from unit 0 to units 3,4, and 5 are bias weights (i.e, the weights represent the thresholds of units 3, 4, and 5). Represented this way, it permits the network to learn suitable thresholds as well. So there are 9 connection weights total, including the 3 threshold weights. Let y_i and α_i respectively denote the output and weighted sum for unit i.

The input and output units of the network are linear units. The hidden units might employ the *binary* squashing function (or the logistic function):

$$y_i = f(\alpha_i) = 1/(1+e^{-\alpha}) \qquad \in (0,1) \tag{8.24}$$

with derivative:

$$f'(\alpha) = f(\alpha)(1-f(\alpha)). \tag{8.25}$$

But the experts say that we should get better learning performance using the *bipolar* squashing function:

$$y_i = g(\alpha_i) = 2f(\alpha) - 1 = (1-e^{-\alpha})/(1+e^{-\alpha}) \qquad \in (-1,1) \tag{8.26}$$

and:

$$g'(\alpha) = 0.5(1+g(\alpha))(1-g(\alpha)). \tag{8.27}$$

Specializing the general algorithm to this network, and using the bipolar squashing function and linear output unit, one obtains the following:

$$unit\ 5\ (linear): \quad y_5 = \alpha_5 \tag{8.28}$$

$$units\ 3,4\ (bipolar) \quad y_i = g(\alpha_i) = (1-e^{-\alpha})/(1+e^{-\alpha}) \tag{8.29}$$

$$\tag{8.30}$$

where

$$\alpha_i = \sum_j w_{j,i} y_j .$$

The output values y_1 and y_2 are the input values, q_1 and q_2, respectively; each is either -1 or 1, and y_0 is always 1.

Here is the weight-update rule for Unit 5: for $i = 0, 3, 4$ (the numbers of the units that provide Unit 5's input):

$$w_{i,5}(t+1) = w_{i,5}(t) + \eta[d(t) - y_5(t)]y_i(t),$$

where $d(t)$ is the desired response at step t. We write this as:

$$w_{i,5}(t+1) = w_{i,5}(t) + \eta\delta_5(t)y_i.$$

For Unit 4: for $i = 0, 1, 2$:

$$w_{i,4}(t+1) = w_{i,4}(t) + \eta\delta_4(t)y_i,$$

and

$$\delta_4(t) = \delta_5(t)w_{4,5}(t)(1 + y_4(t))(1 - y_4(t))$$

where I have lumped the 0.5 into the learning rate, η. This looks a bit different from the usual backpropagation rule because you are using the bipolar squashing function (with asymptotes at -1 and $+1$ instead of 0 and 1. You can figure out the equation for Unit 3's weights. Units 0, 1, and 2 do not have any input weights.

There are only four training patterns: $(-1, -1)$, $(-1, 1)$, $(1, -1)$, and $(1, 1)$ with the respective desired responses -1, 1, 1, and -1 (recall, the output unit is linear, not logistic). You could use these desired responses directly, or you could try the following for the output unit: if the actual output has the correct sign and its magnitude is greater than some predetermined value, such as 1, call the error zero; otherwise use the usual error. Does this help?

Pick training instances randomly, i.e., at each step pick one of the four with probability 0.25. Update the weights after each presentation. Start with initial weights chosen randomly from the interval $[-0.5, 0.5]$. Try $\eta = 0.05$. You may want to try other values for η to see if it makes much difference. You might also want to try momentum on this problem. Keep a running average of the squared error using a window of K time steps, where K is, say 50. You can do this by saving the last K squared errors, and at each time step, sum them and divide by K (or you can use a more clever scheme). Train the network until this running average decreases to 0.01. Give at least one learning curve (the running average of the squared error versus time). Did your network ever get stuck in a local minimum? Describe any other insights that your experiences with this task provided.