Example 2.12 A Perceptron for the AND function: bipolar inputs and targets

The training process for bipolar input, $\alpha = 1$, and threshold and initial weights = 0 is:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
(x_1	x_2	1)							(w_1	w_2	b)
									(0	0	0)
(1	1	1)	0	0	1	(1	1	1)	(1	1	1)
(1	−1	1)	1	1	−1	(−1	1	−1)	(0	2	0)
(−1	1	1)	2	1	−1	(1	−1	−1)	(1	1	−1)
(−1	−1	1)	−3	−1	−1	(0	0	0)	(1	1	−1)

In the second epoch of training, we have:

(1	1	1)	1	1	1	(0	0	0)	(1	1	−1)
(1	−1	1)	−1	−1	−1	(0	0	0)	(1	1	−1)
(−1	1	1)	−1	−1	−1	(0	0	0)	(1	1	−1)
(−1	−1	1)	−3	−1	−1	(0	0	0)	(1	1	−1)

Since all the Δw's are 0 in epoch 2, the system was fully trained after the first epoch.

It seems intuitively obvious that a procedure that could continue to learn to improve its weights even after the classifications are all correct would be better than a learning rule in which weight updates cease as soon as all training patterns are classified correctly. However, the foregoing example shows that the change from binary to bipolar representation improves the results rather spectacularly.

We next show that the perceptron with $\alpha = 1$ and $\theta = .1$ can solve the problem the Hebb rule could not.

Other simple examples

Example 2.13 Perceptron training is more powerful than Hebb rule training

The mapping of interest maps the first three components of the input vector onto a target value that is 1 if there are no zero inputs and that is −1 if there is one zero input. (If there are two or three zeros in the input, we do not specify the target value.) This is a portion of the parity problem for three inputs. The fourth component of the input vector is the input to the bias weight and is therefore always 1. The weight change vector is left blank if no error has occurred for a particular pattern. The learning rate is $\alpha = 1$, and the threshold $\theta = .1$. We show the following selected epochs:

INPUT	NET	OUT	TARGET	WEIGHT CHANGE	WEIGHTS
x_1 x_2 x_3 1					(w_1 w_2 w_3 b)
					(0 0 0 0)
Epoch 1:					
(1 1 1 1)	0	0	1	(1 1 1 1)	(1 1 1 1)
(1 1 0 1)	3	1	−1	(−1 −1 0 −1)	(0 0 1 0)
(1 0 1 1)	1	1	−1	(−1 0 −1 −1)	(−1 0 0 −1)
(0 1 1 1)	−1	−1	−1	( )	(−1 0 0 −1)
Epoch 2:					
(1 1 1 1)	−2	−1	1	(1 1 1 1)	(0 1 1 0)
(1 1 0 1)	1	1	−1	(−1 −1 0 −1)	(−1 0 1 −1)
(1 0 1 1)	−1	−1	−1	( )	(−1 0 1 −1)
(0 1 1 1)	0	0	−1	(0 −1 −1 −1)	(−1 −1 0 −2)
Epoch 3:					
(1 1 1 1)	−4	−1	1	(1 1 1 1)	(0 0 1 −1)
(1 1 0 1)	−1	−1	−1	( )	(0 0 1 −1)
(1 0 1 1)	0	0	−1	(−1 0 −1 −1)	(−1 0 0 −2)
(0 1 1 1)	−2	−1	−1	( )	(−1 0 0 −2)
Epoch 4:					
(1 1 1 1)	−3	−1	1	(1 1 1 1)	(0 1 1 −1)
(1 1 0 1)	0	0	−1	(−1 −1 0 −1)	(−1 0 1 −2)
(1 0 1 1)	−2	−1	−1	( )	(−1 0 1 −2)
(0 1 1 1)	−1	−1	−1	( )	(−1 0 1 −2)
Epoch 5:					
(1 1 1 1)	−2	−1	1	(1 1 1 1)	(0 1 2 −1)
(1 1 0 1)	0	0	−1	(−1 −1 0 −1)	(−1 0 2 −2)
(1 0 1 1)	−1	−1	−1	( )	(−1 0 2 −2)
(0 1 1 1)	0	0	−1	(0 −1 −1 −1)	(−1 −1 1 −3)
Epoch 10:					
(1 1 1 1)	−3	−1	1	(1 1 1 1)	(1 1 2 −3)
(1 1 0 1)	−1	−1	−1	( )	(1 1 2 −3)
(1 0 1 1)	0	0	−1	(−1 0 −1 −1)	(0 1 1 −4)
(0 1 1 1)	−2	−1	−1	( )	(0 1 1 −4)

Epoch 15:

(1	1	1	1)	−1	−1	1	(1	1	1	1)	(1	3	3	−4)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(0	2	3	−5)
(1	0	1	1)	−2	−1	−1	(			)	(0	2	3	−5)
(0	1	1	1)	0	0	−1	(0	−1	−1	−1)	(0	1	2	−6)

Epoch 20:

(1	1	1	1)	−2	−1	1	(1	1	1	1)	(2	2	4	−6)
(1	1	0	1)	−2	−1	−1	(			)	(2	2	4	−6)
(1	0	1	1)	0	0	−1	(−1	0	−1	−1)	(1	2	3	−7)
(0	1	1	1)	−2	−1	−1	(			)	(1	2	3	−7)

Epoch 25:

(1	1	1	1)	0	0	1	(1	1	1	1)	(3	4	4	−7)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(2	3	4	−8)
(1	0	1	1)	−2	−1	−1	(			)	(2	3	4	−8)
(0	1	1	1)	−1	−1	−1	(			)	(2	3	4	−8)

Epoch 26:

(1	1	1	1)	1	1	1	(			)	(2	3	4	−8)
(1	1	0	1)	−3	−1	−1	(			)	(2	3	4	−8)
(1	0	1	1)	−2	−1	−1	(			)	(2	3	4	−8)
(0	1	1	1)	−1	−1	−1	(			)	(2	3	4	−8)

Character recognition

Application Procedure

Step 0. Apply training algorithm to set the weights.

Step 1. For each input vector **x** to be classified, do Steps 2–3.

Step 2. Set activations of input units.

Step 3. Compute response of output unit:

$$y\_in = \sum_i x_i w_i;$$

$$y = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \le y\_in \le \theta \\ -1 & \text{if } y\_in < -\theta \end{cases}$$

Example 2.14 A Perceptron to classify letters from different fonts: one output class

As the first example of using the perceptron for character recognition, consider the 21 input patterns in Figure 2.20 as examples of *A* or not-*A*. In other words, we train the perceptron to classify each of these vectors as belonging, or not belonging, to

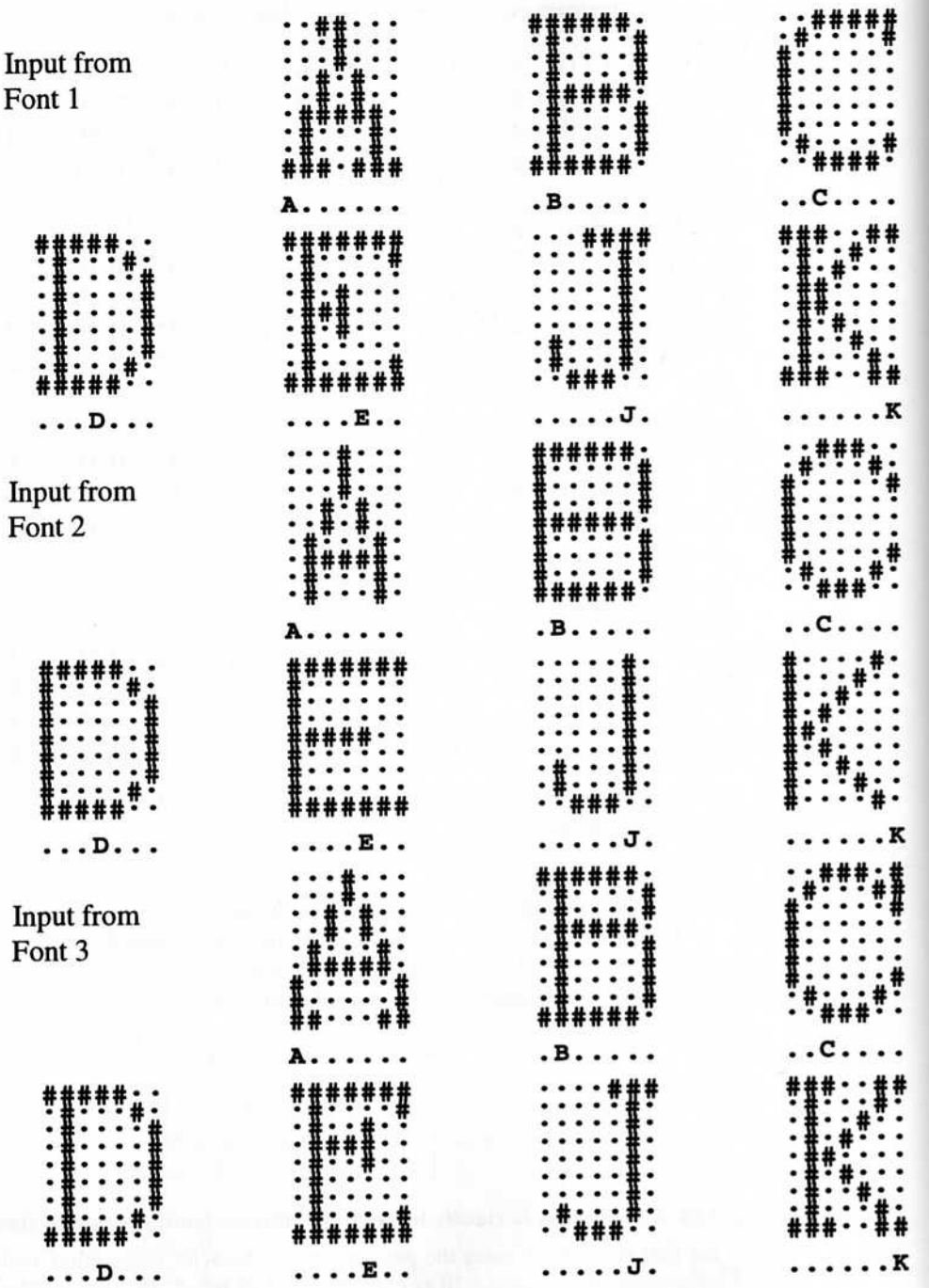

Figure 2.20 Training input and target output patterns.

the class A. In that case, the target value for each pattern is either 1 or -1; only the first component of the target vector shown is applicable. The net is as shown in Figure 2.14, and $n = 63$. There are three examples of A and 18 examples of not-A in Figure 2.20.

We could, of course, use the same vectors as examples of B or not-B and train the net in a similar manner. Note, however, that because we are using a single-layer net, the weights for the output unit signifying A do not have any interaction with the weights for the output unit signifying B. Therefore, we can solve these two problems at the same time, by allowing a column of weights for each output unit. Our net would have 63 input units and 2 output units. The first output unit would correspond to "A or not-A", the second unit to "B or not-B." Continuing this idea, we can identify 7 output units, one for each of the 7 categories into which we wish to classify our input.

Ideally, when an unknown character is presented to the net, the net's output consists of a single "yes" and six "nos." In practice, that may not happen, but the net may produce several guesses that can be resolved by other methods, such as considering the strengths of the activations of the various output units prior to setting the threshold or examining the context in which the ill-classified character occurs.

Example 2.15 A Perceptron to classify letters from different fonts: several output classes

The perceptron shown in Figure 2.14 can be extended easily to the case where the input vectors belong to one (or more) of several categories. In this type of application, there is an output unit representing each of the categories to which the input vectors may belong. The architecture of such a net is shown in Figure 2.21.

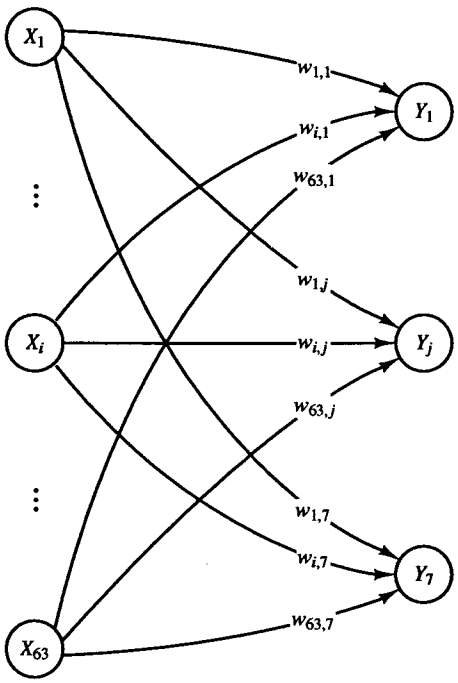

Figure 2.21 Perceptron to classify input into seven categories.

For this example, each input vector is a 63-tuple representing a letter expressed as a pattern on a 7 × 9 grid of pixels. The training patterns are illustrated in Figure 2.20. There are seven categories to which each input vector may belong, so there are seven components to the output vector, each representing a letter: A, B, C, D, E, K, or J. For ease of reading, we show the target output pattern indicating that the input was an "A" as (A · · · · · · ·), a "B" (· B · · · · · ·), etc.

The training input patterns and target responses must be converted to an appropriate form for the neural net to process. A bipolar representation has better computational characteristics than does a binary representation. The input patterns may be converted to bipolar vectors as described in Example 2.8; the target output pattern (A · · · · · · ·) becomes the bipolar vector (1, −1, −1, −1, −1, −1, −1) and the target pattern (· B · · · · · ·) is represented by the bipolar vector (−1, 1, −1, −1, −1, −1, −1).

A modified training algorithm for several output categories (threshold = 0, learning rate = 1, bipolar training pairs) is as follows:

Step 0. Initialize weights and biases
(0 or small random values).

Step 1. While stopping condition is false, do Steps 1–6.

Step 2. For each bipolar training pair **s : t**, do Steps 3–5.

Step 3. Set activation of each input unit, $i = 1, \ldots, n$:

$$x_i = s_i.$$

Step 4. Compute activation of each output unit, $j = 1, \ldots, m$:

$$y\_in_j = b_j + \sum_i x_i w_{ij}.$$

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > \theta \\ 0 & \text{if } -\theta \le y\_in_j \le \theta \\ -1 & \text{if } y\_in_j < -\theta \end{cases}$$

Step 5. Update biases and weights, $j = 1, \ldots, m$; $i = 1, \ldots, n$:
If $t_j \neq y_j$, then

$$b_j(\text{new}) = b_j(\text{old}) + t_j;$$

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + t_j x_i.$$

Else, biases and weights remain unchanged.

Step 6. Test for stopping condition:
If no weight changes occurred in Step 2, stop; otherwise, continue.

After training, the net correctly classifies each of the training vectors.

The performance of the net shown in Figure 2.21 in classifying input vectors that are similar to the training vectors is shown in Figure 2.22. Each of the input

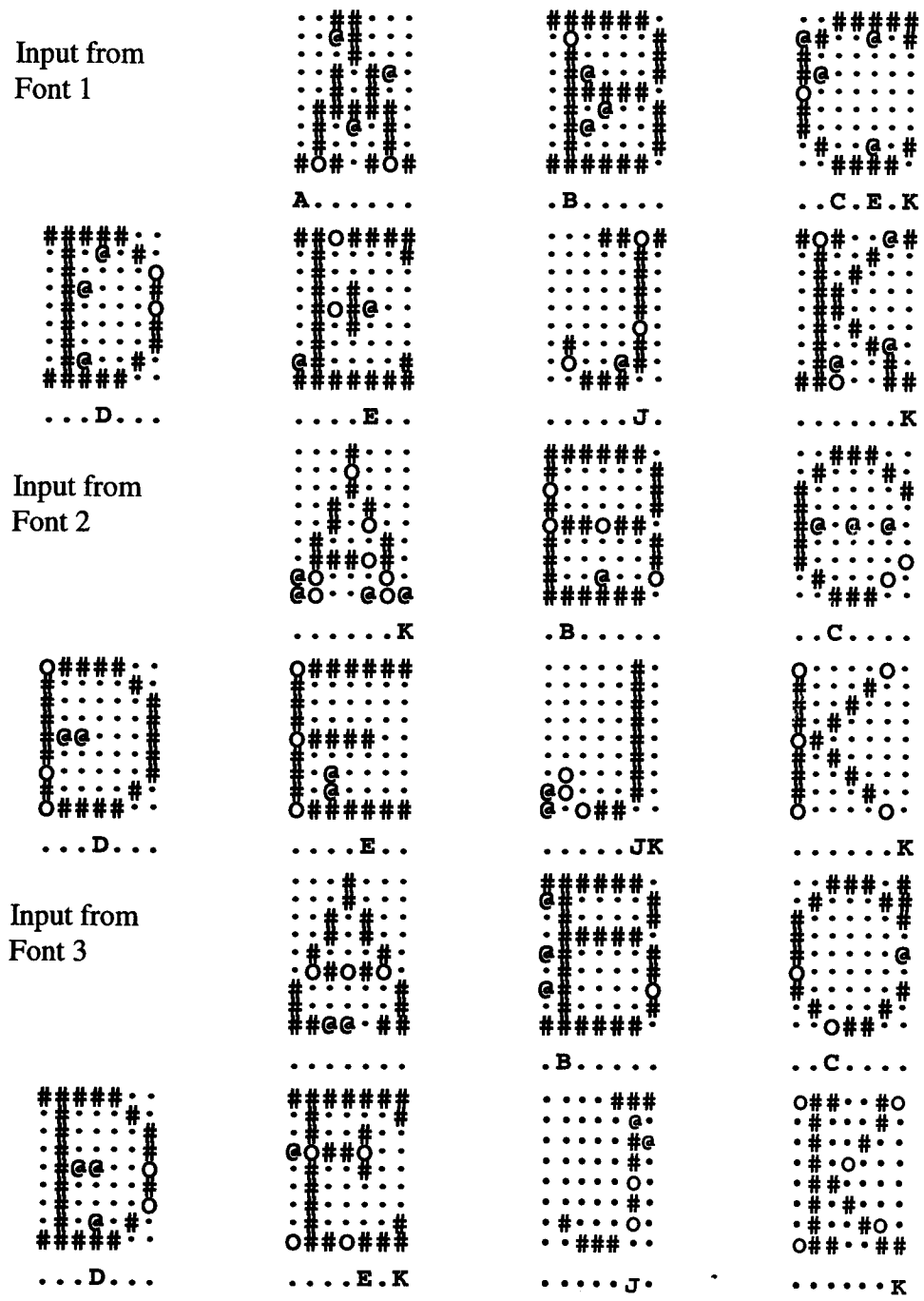

Figure 2.22 Classification of noisy input patterns using a perceptron.

patterns is a training input pattern with a few of its pixels changed. The pixels where the input pattern differs from the training pattern are indicated by @ for a pixel that is "on" now but was "off" in the training pattern, and O for a pixel that is "off" now but was originally "on."

2.3.4 Perceptron Learning Rule Convergence Theorem

The statement and proof of the perceptron learning rule convergence theorem given here are similar to those presented in several other sources [Hertz, Krogh, & Palmer, 1991; Minsky & Papert, 1988; Arbib, 1987]. Each of these provides a slightly different perspective and insights into the essential aspects of the rule. The fact that the weight vector is perpendicular to the plane separating the input patterns at each step of the learning processes [Hertz, Krogh, & Palmer, 1991] can be used to interpret the degree of difficulty of training a perceptron for different types of input.

The perceptron learning rule is as follows:

Given a finite set of P input training vectors

$$\mathbf{x}(p), \quad p = 1, \ldots, P,$$

each with an associated target value

$$t(p), \quad p = 1, \ldots, P,$$

which is either $+1$ or -1, and an activation function $y = f(y\_in)$, where

$$y = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \leq y\_in \leq \theta \\ -1 & \text{if } y\_in < -\theta, \end{cases}$$

the weights are updated as follows:

If $y \neq t$, then

$$\mathbf{w} \text{ (new)} = \mathbf{w} \text{ (old)} + t\mathbf{x};$$

else

no change in the weights.

The perceptron learning rule convergence theorem is:

If there is a weight vector $\mathbf{w}^*$ such that $f(\mathbf{x}(p) \cdot \mathbf{w}^*) = t(p)$ for all p, then for any starting vector $\mathbf{w}$, the perceptron learning rule will converge to a weight vector (not necessarily unique and not necessarily $\mathbf{w}^*$) that gives the correct response for all training patterns, and it will do so in a finite number of steps.

The proof of the theorem is simplified by the observation that the training set can be considered to consist of two parts:

$$F^+ = \{\mathbf{x} \text{ such that the target value is } +1\}$$

and

$$F^- = \{\mathbf{x} \text{ such that the target value is } -1\}.$$

A new training set is then defined as

$$F = F^+ \cup -F^-,$$

where

$$-F^- = \{-\mathbf{x} \text{ such that } \mathbf{x} \text{ is in } F^-\}.$$

In order to simplify the algebra slightly, we shall assume, without loss of generality, that $\theta = 0$ and $\alpha = 1$ in the proof. The existence of a solution of the original problem, namely the existence of a weight vector $\mathbf{w}^*$ for which

$$\mathbf{x}\cdot\mathbf{w}^* > 0 \qquad \text{if } \mathbf{x} \text{ is in } F^+$$

and

$$\mathbf{x}\cdot\mathbf{w}^* < 0 \qquad \text{if } \mathbf{x} \text{ is in } F^-,$$

is equivalent to the existence of a weight vector $\mathbf{w}^*$ for which

$$\mathbf{x}\cdot\mathbf{w}^* > 0 \qquad \text{if } \mathbf{x} \text{ is in } F.$$

All target values for the modified training set are $+1$. If the response of the net is incorrect for a given training input, the weights are updated according to

$$\mathbf{w}(\text{new}) = \mathbf{w}(\text{old}) + \mathbf{x}.$$

Note that the input training vectors must each have an additional component (which is always 1) included to account for the signal to the bias weight.

We now sketch the proof of this remarkable convergence theorem, because of the light that it sheds on the wide variety of forms of perceptron learning that are guaranteed to converge. As mentioned, we assume that the training set has been modified so that all targets are $+1$. Note that this will involve reversing the sign of all components (including the input component corresponding to the bias) for any input vectors for which the target was originally -1.

We now consider the sequence of input training vectors for which a weight change occurs. We must show that this sequence is finite.

Let the starting weights be denoted by $\mathbf{w}(0)$, the first new weights by $\mathbf{w}(1)$, etc. If $\mathbf{x}(0)$ is the first training vector for which an error has occurred, then

$$\mathbf{w}(1) = \mathbf{w}(0) + \mathbf{x}(0) \qquad (\text{where, by assumption, } \mathbf{x}(0)\cdot\mathbf{w}(0) \leq 0).$$

If another error occurs, we denote the vector $\mathbf{x}(1)$; $\mathbf{x}(1)$ may be the same as $\mathbf{x}(0)$ if no errors have occurred for any other training vectors, or $\mathbf{x}(1)$ may be different from $\mathbf{x}(0)$. In either case,

$$\mathbf{w}(2) = \mathbf{w}(1) + \mathbf{x}(1) \qquad (\text{where, by assumption, } \mathbf{x}(1)\cdot\mathbf{w}(1) \leq 0).$$

At any stage, say, k, of the process, the weights are changed if and only if the current weights fail to produce the correct (positive) response for the current input vector, i.e., if $\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) \leq 0$. Combining the successive weight changes gives

$$\mathbf{w}(k) = \mathbf{w}(0) + \mathbf{x}(0) + \mathbf{x}(1) + \mathbf{x}(2) + \cdots + \mathbf{x}(k-1).$$

We now show that k cannot be arbitrarily large.

Let $\mathbf{w}^*$ be a weight vector such that $\mathbf{x}\cdot\mathbf{w}^* > 0$ for all training vectors in F. Let $m = \min\{\mathbf{x}\cdot\mathbf{w}^*\}$, where the minimum is taken over all training vectors in F; this minimum exists as long as there are only finitely many training vectors. Now,

$$\begin{aligned}\mathbf{w}(k)\cdot\mathbf{w}^* &= [\mathbf{w}(0) + \mathbf{x}(0) + \mathbf{x}(1) + \mathbf{x}(2) + \cdots + \mathbf{x}(k-1)]\cdot\mathbf{w}^* \\ &\geq \mathbf{w}(0)\cdot\mathbf{w}^* + km\end{aligned}$$

since $\mathbf{x}(i)\cdot\mathbf{w}^* \geq m$ for each i, $1 \leq i \leq P$.

The Cauchy-Schwartz inequality states that for any vectors $\mathbf{a}$ and $\mathbf{b}$,

$$(\mathbf{a}\cdot\mathbf{b})^2 \leq \|\mathbf{a}\|^2 \|\mathbf{b}\|^2,$$

or

$$\|\mathbf{a}\|^2 \geq \frac{(\mathbf{a}\cdot\mathbf{b})^2}{\|\mathbf{b}\|^2} \qquad (\text{for } \|\mathbf{b}\|^2 \neq 0).$$

Therefore,

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &\geq \frac{(\mathbf{w}(k)\cdot\mathbf{w}^*)^2}{\|\mathbf{w}^*\|^2} \\ &\geq \frac{(\mathbf{w}(0)\cdot\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2}.\end{aligned}$$

This shows that the squared length of the weight vector grows faster than k^2, where k is the number of time the weights have changed.

However, to show that the length cannot continue to grow indefinitely, consider

$$\mathbf{w}(k) = \mathbf{w}(k-1) + \mathbf{x}(k-1),$$

together with the fact that

$$\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) \leq 0.$$

By simple algebra,

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &= \|\mathbf{w}(k-1)\|^2 + 2\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(k-1)\|^2 + \|\mathbf{x}(k-1)\|^2.\end{aligned}$$

Now let $M = \max \{\| \mathbf{x} \|^2 \text{ for all } \mathbf{x} \text{ in the training set}\}$; then

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &\leq \|\mathbf{w}(k-1)\|^2 + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(k-2)\|^2 + \|\mathbf{x}(k-2)\|^2 + \|\mathbf{x}(k-1)\|^2 \\ &\vdots \\ &\leq \|\mathbf{w}(0)\|^2 + \|\mathbf{x}(0)\|^2 + \cdots + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(0)\|^2 + kM.\end{aligned}$$

Thus, the squared length grows less rapidly than linearly in k.

Combining the inequalities

$$\|\mathbf{w}(k)\|^2 \geq \frac{(\mathbf{w}(0)\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2}$$

and

$$\|\mathbf{w}(k)\|^2 \leq \|\mathbf{w}(0)\|^2 + kM$$

shows that the number of times that the weights may change is bounded. Specifically,

$$\frac{(\mathbf{w}(0)\cdot\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2} \leq \|\mathbf{w}(k)\|^2 \leq \|\mathbf{w}(0)\|^2 + kM.$$

Again, to simplify the algebra, assume (without loss of generality) that $\mathbf{w}(0) = 0$. Then the maximum possible number of times the weights may change is given by

$$\frac{(km)^2}{\|\mathbf{w}^*\|^2} \leq kM,$$

or

$$k \leq \frac{M\,\|\mathbf{w}^*\|^2}{m^2}.$$

Since the assumption that $\mathbf{w}^*$ exists can be restated, without loss of generality, as the assumption that there is a solution weight vector of unit length (and the definition of m is modified accordingly), the maximum number of weight updates is M/m^2. Note, however, that many more computations may be required, since very few input vectors may generate an error during any one epoch of training. Also, since $\mathbf{w}^*$ is unknown (and therefore, so is m), the number of weight updates cannot be predicted from the preceding inequality.

The foregoing proof shows that many variations in the perceptron learning rule are possible. Several of these variations are explicitly mentioned in Chapter 11 of Minsky and Papert (1988).

The original restriction that the coefficients of the patterns be binary is un-

necessary. All that is required is that there be a finite maximum norm of the training vectors (or at least a finite upper bound to the norm). Training may take a long time (a large number of steps) if there are training vectors that are very small in norm, since this would cause small m to have a small value. The argument of the proof is unchanged if a nonzero value of θ is used (although changing the value of θ may change a problem from solvable to unsolvable or vice versa). Also, the use of a learning rate other than 1 will not change the basic argument of the proof (see Exercise 2.8). Note that there is no requirement that there can be only finitely many training vectors, as long as the norm of the training vectors is bounded (and bounded away from 0 as well). The actual target values do not matter, either; the learning law simply requires that the weights be incremented by the input vector (or a multiple of it) whenever the response of the net is incorrect (and that the training vectors can be stated in such a way that they all should give the same response of the net).

Variations on the learning step include setting the learning rate α to any nonnegative constant (Minsky starts by setting it specifically to 1), setting α to $1/\|\mathbf{x}\|$ so that the weight change is a unit vector, and setting α to $(\mathbf{x} \cdot \mathbf{w})/\|\mathbf{x}\|^2$ (which makes the weight change just enough for the pattern $\mathbf{x}$ to be classified correctly at this step).

Minsky sets the initial weights equal to an arbitrary training pattern. Others usually indicate small random values.

Note also that since the procedure will converge from an arbitrary starting set of weights, the process is error correcting, as long as the errors do not occur too often (and the process is not stopped before error correction occurs).

2.4 ADALINE

The ADALINE (ADAptive Linear NEuron) [Widrow & Hoff, 1960] typically uses bipolar (1 or -1) activations for its input signals and its target output (although it is not restricted to such values). The weights on the connections from the input units to the ADALINE are adjustable. The ADALINE also has a bias, which acts like an adjustable weight on a connection from a unit whose activation is always 1.

In general, an ADALINE can be trained using the delta rule, also known as the least mean squares (LMS) or Widrow-Hoff rule. The rule (Section 2.4.2) can also be used for single-layer nets with several output units; an ADALINE is a special case in which there is only one output unit. During training, the activation of the unit is its net input, i.e., the activation function is the identity function. The learning rule minimizes the mean squared error between the activation and the target value. This allows the net to continue learning on all training patterns, even after the correct output value is generated (if a threshold function is applied) for some patterns.

After training, if the net is being used for pattern classification in which the desired output is either a $+1$ or a -1, a threshold function is applied to the net

input to obtain the activation. If the net input to the ADALINE is greater than or equal to 0, then its activation is set to 1; otherwise it is set to −1. Any problem for which the input patterns corresponding to the output value +1 are linearly separable from input patterns corresponding to the output value −1 can be modeled successfully by an ADALINE unit. An application algorithm is given in Section 2.4.3 to illustrate the use of the activation function after the net is trained.

In Section 2.4.4, we shall see how a heuristic learning rule can be used to train a multilayer combination of ADALINES, known as a MADALINE.

2.4.1 Architecture

An ADALINE is a single unit (neuron) that receives input from several units. It also receives input from a "unit" whose signal is always +1, in order for the bias weight to be trained by the same process (the delta rule) as is used to train the other weights. A single ADALINE is shown in Figure 2.23.

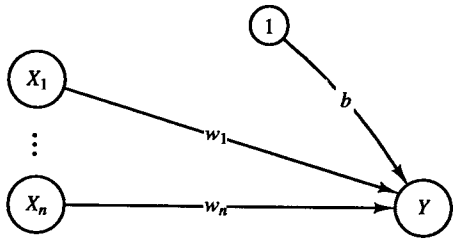

Figure 2.23 Architecture of an ADALINE.

Several ADALINES that receive signals from the same input units can be combined in a single-layer net, as described for the perceptron (Section 2.3.3). If, however, ADALINES are combined so that the output from some of them becomes input for others of them, then the net becomes multilayer, and determining the weights is more difficult. Such a multilayer net, known as a MADALINE, is considered in Section 2.4.5.

2.4.2 Algorithm

A training algorithm for an ADALINE is as follows:

Step 0. Initialize weights.
(Small random values are usually used.)
Set learning rate α.
(See comments following algorithm.)

Step 1. While stopping condition is false, do Steps 2–6.

Step 2. For each bipolar training pair s:t, do Steps 3–5.

Step 3. Set activations of input units, $i = 1, \ldots, n$:

$$x_i = s_i.$$

Step 4. Compute net input to output unit:

$$y\_in = b + \sum_i x_i w_i.$$

Step 5. Update bias and weights, $i = 1, \ldots, n$:

$$b(\text{new}) = b(\text{old}) + \alpha(t - y\_in).$$

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y\_in)x_i.$$

Step 6. Test for stopping condition:
If the largest weight change that occurred in Step 2 is smaller than a specified tolerance, then stop; otherwise continue.

Setting the learning rate to a suitable value requires some care. According to Hecht-Nielsen (1990), an upper bound for its value can be found from the largest eigenvalue of the correlation matrix R of the input (row) vectors $\mathbf{x}(p)$:

$$R = \frac{1}{P} \sum_{p=1}^{P} \mathbf{x}(p)^T \mathbf{x}(p),$$

namely,

$$\alpha < \text{one-half the largest eigenvalue of } R.$$

However, since R does not need to be calculated to compute the weight updates, it is common simply to take a small value for α (such as $\alpha = .1$) initially. If too large a value is chosen, the learning process will not converge; if too small a value is chosen, learning will be extremely slow [Hecht-Nielsen, 1990]. The choice of learning rate and methods of modifying it are considered further in Chapter 6. For a single neuron, a practical range for the learning rate α is $0.1 \leq n\alpha \leq 1.0$, where n is the number of input units [Widrow, Winter & Baxter, 1988].

The proof of the convergence of the ADALINE training process is essentially contained in the derivation of the delta rule, which is given in Section 2.4.4.

2.4.3 Applications

After training, an ADALINE unit can be used to classify input patterns. If the target values are bivalent (binary or bipolar), a step function can be applied as the

activation function for the output unit. The following procedure shows the step function for bipolar targets, the most common case:

Step 0. Initialize weights
(from ADALINE training algorithm given in Section 2.4.2).

Step 1. For each bipolar input vector **x**, do Steps 2–4.

Step 2. Set activations of the input units to **x**.

Step 3. Compute net input to output unit:

$$y\_in = b + \sum_i x_i w_i.$$

Step 4. Apply the activation function:

$$y = \begin{cases} 1 & \text{if } y\_in \geq 0; \\ -1 & \text{if } y\_in < 0. \end{cases}$$

Simple examples

The weights (and biases) in Examples 2.16–2.19 give the minimum total squared error for each set of training patterns. Good approximations to these values can be found using the algorithm in Section 2.4.2 with a small learning rate.

Example 2.16 An ADALINE for the AND function: binary inputs, bipolar targets

Even though the ADALINE was presented originally for bipolar inputs and targets, the delta rule also applies to binary input. In this example, we consider the AND function with binary input and bipolar targets. The function is defined by the following four training patterns:

x_1	x_2	t
1	1	1
1	0	−1
0	1	−1
0	0	−1

As indicated in the derivation of the delta rule (Section 2.4.4), an ADALINE is designed to find weights that minimize the total error

$$E = \sum_{p=1}^{4} (x_1(p)w_1 + x_2(p)w_2 + w_0 - t(p))^2,$$

where

$$x_1(p)w_1 + x_2(p)w_2 + w_0$$

is the net input to the output unit for pattern p and $t(p)$ is the associated target for pattern p.

Weights that minimize this error are

$$w_1 = 1$$

and

$$w_2 = 1,$$

with the bias

$$w_0 = -\frac{3}{2}.$$

Thus, the separating line is

$$x_1 + x_2 - \frac{3}{2} = 0.$$

The total squared error for the four training patterns with these weights is 1.

A minor modification to Example 2.11 (setting $\theta = 0$) shows that for the perceptron, the boundary line is

$$x_2 = -\frac{2}{3}x_1 + \frac{4}{3}.$$

(The two boundary lines coincide when $\theta = 0$.) The total squared error for the minimizing weights found by the perceptron is 10/9.

Example 2.17 An ADALINE for the AND function: bipolar inputs and targets

The weights that minimize the total error for the bipolar form of the AND function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = \frac{1}{2},$$

with the bias

$$w_0 = -\frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{1}{2} = 0,$$

which is of course the same line as

$$x_1 + x_2 - 1 = 0,$$

as found by the perceptron in Example 2.12.

Example 2.18 An ADALINE for the AND NOT function: bipolar inputs and targets

The logic function x_1 AND NOT x_2 is defined by the following bipolar input and target patterns:

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

Weights that minimize the total squared error for the bipolar form of the AND NOT function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = -\frac{1}{2},$$

with the bias

$$w_0 = -\frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 - \frac{1}{2}x_2 - \frac{1}{2} = 0.$$

Example 2.19 An ADALINE for the OR function: bipolar inputs and targets

The logic function x_1 OR x_2 is defined by the following bipolar input and target patterns:

x_1	x_2	t
1	1	1
1	−1	1
−1	1	1
−1	−1	−1

Weights that minimize the total squared error for the bipolar form of the OR function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = \frac{1}{2},$$

with the bias

$$w_0 = \frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 + \frac{1}{2} = 0.$$

2.4.4 Derivations

Delta rule for single output unit

The delta rule changes the weights of the neural connections so as to minimize the difference between the net input to the output unit, *y_in*, and the target value *t*. The aim is to minimize the error over all training patterns. However, this is accomplished by reducing the error for each pattern, one at a time. Weight corrections can also be accumulated over a number of training patterns (so-called batch updating) if desired. In order to distinguish between the fixed (but arbitrary) index for the weight whose adjustment is being determined in the derivation that follows and the index of summation needed in the derivation, we use the index *I* for the weight and the index *i* for the summation. We shall return to the more standard lowercase indices for weights whenever this distinction is not needed. The delta rule for adjusting the *I*th weight (for each pattern) is

$$\Delta w_I = \alpha(t - y\_in)x_I.$$

The nomenclature we use in the derivation is as follows:

x vector of activations of input units, an *n*-tuple.

y_in the net input to output unit *Y* is

$$y\_in = \sum_{i=1}^{n} x_i w_i.$$

t target output.

Derivation. The squared error for a particular training pattern is

$$E = (t - y\_in)^2.$$

E is a function of all of the weights, w_i, $i = 1, \ldots, n$. The gradient of *E* is the vector consisting of the partial derivatives of *E* with respect to each of the weights. The gradient gives the direction of most rapid increase in *E*; the opposite direction

gives the most rapid decrease in the error. The error can be reduced by adjusting the weight w_I in the direction of $-\dfrac{\partial E}{\partial w_I}$.

Since $y\_in = \sum_{i=1}^{n} x_i w_i$,

$$\frac{\partial E}{\partial w_I} = -2(t - y\_in)\frac{\partial y\_in}{\partial w_I}$$

$$= -2(t - y\_in)x_I.$$

Thus, the local error will be reduced most rapidly (for a given learning rate) by adjusting the weights according to the delta rule,

$$\Delta w_I = \alpha(t - y\_in)x_I.$$

Delta rule for several output units

The derivation given in this subsection allows for more than one output unit. The weights are changed to reduce the difference between the net input to the output unit, y_in_J, and the target value t_J. This formulation reduces the error for each pattern. Weight corrections can also be accumulated over a number of training patterns (so-called batch updating) if desired.

The delta rule for adjusting the weight from the Ith input unit to the Jth output unit (for each pattern) is

$$\Delta w_{IJ} = \alpha(t_J - y\_in_J)x_I.$$

Derivation. The squared error for a particular training pattern is

$$E = \sum_{j=1}^{m} (t_j - y\_in_j)^2.$$

E is a function of all of the weights. The gradient of E is a vector consisting of the partial derivatives of E with respect to each of the weights. This vector gives the direction of most rapid increase in E; the opposite direction gives the direction of most rapid decrease in the error. The error can be reduced most rapidly by adjusting the weight w_{IJ} in the direction of $-\partial E/\partial w_{IJ}$.

We now find an explicit formula for $\partial E/\partial w_{IJ}$ for the arbitrary weight w_{IJ}. First, note that

$$\frac{\partial E}{\partial w_{IJ}} = \frac{\partial}{\partial w_{IJ}} \sum_{j=1}^{m} (t_j - y\_in_j)^2$$

$$= \frac{\partial}{\partial w_{IJ}} (t_J - y\_in_J)^2,$$