# Chapter 4

# Neural Networks

In chapter two we defined several important subsets of Boolean functions. Suppose we decide to use one of these subsets as a hypothesis set for supervised function learning. We next have the question of how best to implement the function as a device that gives the outputs prescribed by the function for arbitrary inputs. In this chapter we describe how networks of non-linear elements can be used to implement various input-output functions and how they can be trained using supervised learning methods.

Networks of non-linear elements, interconnected through adjustable weights, play a prominent role in machine learning. They are called *neural networks* because the non-linear elements have as their inputs a weighted sum of the outputs of other elements—much like networks of biological neurons do. These networks commonly use the threshold element which we encountered in chapter two in our study of linearly separable Boolean functions. We begin our treatment of neural nets by studying this threshold element and how it can be used in the simplest of all networks, namely ones composed of a single threshold element.

## 4.1 Threshold Logic Units

### 4.1.1 Definitions and Geometry

Linearly separable (threshold) functions are implemented in a straightforward way by summing the weighted inputs and comparing this sum to a threshold value as shown in Fig. 4.1. This structure we call a *threshold logic unit (TLU)*. Its output is 1 or 0 depending on whether or not the weighted sum of its inputs is greater than or equal to a threshold value, $\theta$. It has also been called an *Adaline* (for adaptive linear element) [Widrow, 1962, Widrow & Lehr, 1990], an LTU (linear threshold unit), a *perceptron*, and a *neuron*. (Although the word "perceptron" is often used nowadays to refer to a single TLU, Rosenblatt originally defined it as a class of *networks* of threshold elements [Rosenblatt, 1958].)

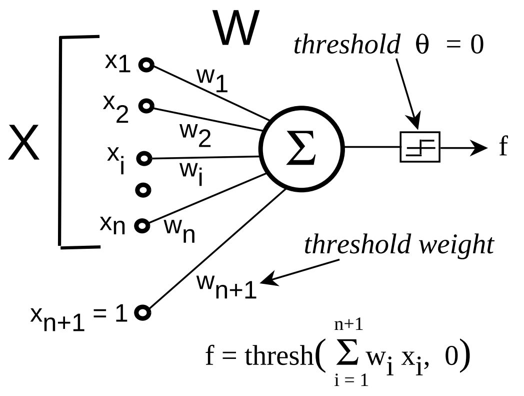


Figure 4.1: A Threshold Logic Unit (TLU)

The $n$-dimensional feature or input vector is denoted by $\mathbf{X} = (x_1, \ldots, x_n)$. When we want to distinguish among different feature vectors, we will attach subscripts, such as $\mathbf{X}_i$. The components of $\mathbf{X}$ can be any real-valued numbers, but we often specialize to the binary numbers 0 and 1. The weights of a TLU are represented by an $n$-dimensional *weight vector*, $\mathbf{W} = (w_1, \ldots, w_n)$. Its components are real-valued numbers (but we sometimes specialize to integers). The TLU has output 1 if $\sum_{i=1}^{n} x_i w_i \geq \theta$; otherwise it has output 0. The weighted sum that is calculated by the TLU can be simply represented as a vector dot product, $\mathbf{X} \bullet \mathbf{W}$. (If the pattern and weight vectors are thought of as "column" vectors, this dot product is then sometimes written as $\mathbf{X}^t\mathbf{W}$, where the "row" vector $\mathbf{X}^t$ is the transpose of $\mathbf{X}$.) Often, the threshold, $\theta$, of the TLU is fixed at 0; in that case, arbitrary thresholds are achieved by using $(n+1)$-dimensional "augmented" vectors, $\mathbf{Y}$, and $\mathbf{V}$, whose first $n$ components are the same as those of $\mathbf{X}$ and $\mathbf{W}$, respectively. The $(n+1)$-st component, $x_{n+1}$, of the augmented feature vector, $\mathbf{Y}$, always has value 1; the $(n+1)$-st component, $w_{n+1}$, of the augmented weight vector, $\mathbf{V}$, is set equal to the negative of the desired threshold value. (When we want to emphasize the use of augmented vectors, we'll use the $\mathbf{Y}$,$\mathbf{V}$ notation; however when the context of the discussion makes it clear about what sort of vectors we are talking about, we'll lapse back into the more familiar $\mathbf{X}$,$\mathbf{W}$ notation.) In the $\mathbf{Y}$,$\mathbf{V}$ notation, the TLU has an output of 1 if $\mathbf{Y}\bullet\mathbf{V} \geq 0$. Otherwise, the output is 0.

We can give an intuitively useful geometric description of a TLU. A TLU divides the input space by a hyperplane as sketched in Fig. 4.2. The hyperplane is the boundary between patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} > 0$ and patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} < 0$. Thus, the equation of the hyperplane itself is $\mathbf{X}\bullet\mathbf{W} + w_{n+1} = 0$. The unit vector that is normal to the hyperplane is $\mathbf{n} = \frac{\mathbf{W}}{|\mathbf{W}|}$, where $|\mathbf{W}| = \sqrt{(w_1^2 + \ldots + w_n^2)}$ is the length of the vector $\mathbf{W}$. (The *normal form* of the hyperplane equation is $\mathbf{X}\bullet\mathbf{n} + \frac{\mathbf{W}}{|\mathbf{W}|} = 0$.) The distance from the hyperplane to the origin is $\frac{w_{n+1}}{|\mathbf{W}|}$, and the distance from an arbitrary point, $\mathbf{X}$, to the hyperplane is $\frac{\mathbf{X}\bullet\mathbf{W} + w_{n+1}}{|\mathbf{W}|}$. When the distance from the hyperplane to the origin is negative (that is, when $w_{n+1} < 0$), then the origin is on the negative side of the hyperplane (that is, the side for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} < 0$).

Adjusting the weight vector, $\mathbf{W}$, changes the orientation of the hyperplane; adjusting $w_{n+1}$ changes the position of the hyperplane (relative to the origin). Thus, training of a TLU can be achieved by adjusting the values of the weights. In this way the hyperplane can be moved so that the TLU implements different (linearly separable) functions of the input.

### 4.1.2 Special Cases of Linearly Separable Functions

#### Terms

Any term of size $k$ can be implemented by a TLU with a weight from each of those inputs corresponding to variables occurring in the term. A weight of $+1$ is used from an input corresponding to a positive literal, and a weight of $-1$ is used from an input corresponding to a negative literal. (Literals not mentioned in the term have weights of zero—that is, no connection at all—from their inputs.) The threshold, $\theta$, is set equal to $k_p - 1/2$, where $k_p$ is the number of positive literals in the term. Such a TLU implements a hyperplane boundary that is parallel to a subface of dimension $(n-k)$ of the unit hypercube. We show a three-dimensional example in Fig. 4.3. Thus, linearly separable functions are a superset of terms.

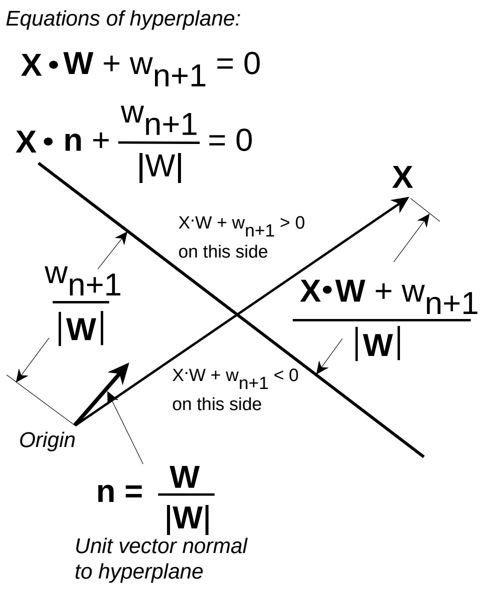

Equations of hyperplane:
$\mathbf{X} \bullet \mathbf{W} + w_{n+1} = 0$
$\mathbf{X} \bullet \mathbf{n} + \frac{w_{n+1}}{|\mathbf{W}|} = 0$
X
$X \cdot W + w_{n+1} > 0$ on this side
$\frac{w_{n+1}}{|\mathbf{W}|}$
$\frac{\mathbf{X} \bullet \mathbf{W} + w_{n+1}}{|\mathbf{W}|}$
$X \cdot W + w_{n+1} < 0$ on this side
Origin
$\mathbf{n} = \frac{\mathbf{W}}{|\mathbf{W}|}$
Unit vector normal to hyperplane

Figure 4.2: TLU Geometry

**Clauses**

The negation of a clause is a term. For example, the negation of the clause $f = x_1 + x_2 + x_3$ is the term $\overline{f} = \overline{x_1}\ \overline{x_2}\ \overline{x_3}$. A hyperplane can be used to implement this term. If we "invert" the hyperplane, it will implement the clause instead. Inverting a hyperplane is done by multiplying all of the TLU weights—even $w_{n+1}$—by $-1$. This process simply changes the orientation of the hyperplane—flipping it around by 180 degrees and thus changing its "positive side." Therefore, linearly separable functions are also a superset of clauses. We show an example in Fig. 4.4.

### 4.1.3 Error-Correction Training of a TLU

There are several procedures that have been proposed for adjusting the weights of a TLU. We present next a family of *incremental* training procedures with parameter $c$. These methods make adjustments to the weight vector only when the TLU being trained makes an error on a training pattern; they are called *error-correction* procedures. We use *augmented* feature and weight vectors in describing them.

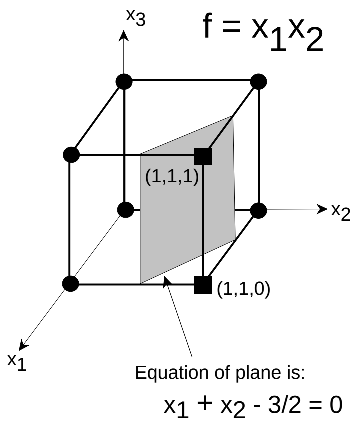


Figure 4.3: Implementing a Term

1. We start with a finite training set, $\Xi$, of vectors, $\mathbf{Y}_i$ , and their binary labels.

2. Compose an infinite training sequence, $\Sigma$, of vectors from $\Xi$ and their labels such that each member of $\Xi$ occurs infinitely often in $\Sigma$. Set the initial weight values of an TLU to arbitrary values.

3. Repeat forever:

   Present the next vector, $\mathbf{Y}_i$, in $\Sigma$ to the TLU and note its response.

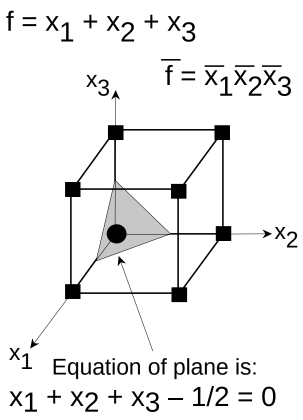
   (a) If the TLU responds correctly, make no change in the weight vector.

   (b) If $\mathbf{Y}_i$ is supposed to produce an output of 0 and produces an output of 1 instead, modify the weight vector as follows:

   $$\mathbf{V} \longleftarrow \mathbf{V} - c_i\mathbf{Y}_i$$

   where $c_i$ is a positive real number called the *learning rate parameter* (whose value is differerent in different instances of this family of procedures and may depend on $i$).

   Note that after this adjustment the new dot product will be $(\mathbf{V} - c_i\mathbf{Y}_i)\bullet\mathbf{Y}_i = \mathbf{V}\bullet\mathbf{Y}_i - c_i\mathbf{Y}_i\bullet\mathbf{Y}_i$, which is smaller than it was before the weight adjustment.



$f = x_1 + x_2 + x_3$
$\overline{f} = \overline{x}_1\overline{x}_2\overline{x}_3$
$x_3$
$x_2$
$x_1$
Equation of plane is:
$x_1 + x_2 + x_3 - 1/2 = 0$

Figure 4.4: Implementing a Clause

(c) If $\mathbf{Y}_i$ is supposed to produce an output of 1 and produces an output of 0 instead, modify the weight vector as follows:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i \mathbf{Y}_i$$

In this case, the new dot product will be $(\mathbf{V} + c_i\mathbf{Y}_i)\bullet\mathbf{Y}_i = \mathbf{V}\bullet\mathbf{Y}_i + c_i\mathbf{Y}_i\bullet\mathbf{Y}_i$, which is larger than it was before the weight adjustment.

Note that all three of these cases can be combined in the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i(d_i - f_i)\mathbf{Y}_i$$

where $d_i$ is the desired response (1 or 0) for $\mathbf{Y}_i$ , and $f_i$ is the actual response (1 or 0) for $\mathbf{Y}_i$.]

Note also that because the weight vector $\mathbf{V}$ now includes the $w_{n+1}$ threshold component, the threshold of the TLU is also changed by these adjustments.

We identify two versions of this procedure:

1) In the *fixed-increment procedure*, the learning rate parameter, $c_i$, is the same fixed, positive constant for all $i$. Depending on the value of this constant, the weight adjustment may or may not correct the response to an erroneously classified feature vector.

2) In the *fractional-correction procedure*, the parameter $c_i$ is set to $\lambda \frac{\mathbf{Y_i} \bullet \mathbf{V}}{\mathbf{Y_i} \bullet \mathbf{Y_i}}$, where **V** is the weight vector *before* it is changed. Note that if $\lambda = 0$, no correction takes place at all. If $\lambda = 1$, the correction is just sufficient to make $\mathbf{Y_i} \bullet \mathbf{V} = 0$. If $\lambda > 1$, the error will be corrected.

It can be proved that if there is some weight vector, **V**, that produces a correct output for all of the feature vectors in $\Xi$, then after a finite number of feature vector presentations, the fixed-increment procedure will find such a weight vector and thus make no more weight changes. The same result holds for the fractional-correction procedure if $1 < \lambda \leq 2$.

For additional background, proofs, and examples of error-correction procedures, see [Nilsson, 1990].

See [Maass & Turán, 1994] for a hyperplane-finding procedure that makes no more than $O(n^2 \log n)$ mistakes.

### 4.1.4 Weight Space

We can give an intuitive idea about how these procedures work by considering what happens to the augmented weight vector in "weight space" as corrections are made. We use augmented vectors in our discussion here so that the threshold function compares the dot product, $\mathbf{Y}_i \bullet \mathbf{V}$, against a threshold of 0. A particular weight vector, **V**, then corresponds to a point in $(n + 1)$-dimensional weight space. Now, for any pattern vector, $\mathbf{Y}_i$, consider the locus of all points in weight space corresponding to weight vectors yielding $\mathbf{Y}_i \bullet \mathbf{V} = 0$. This locus is a hyperplane passing through the origin of the $(n + 1)$-dimensional space. Each pattern vector will have such a hyperplane corresponding to it. Weight points in one of the half-spaces defined by this hyperplane will cause the corresponding pattern to yield a dot product less than 0, and weight points in the other half-space will cause the corresponding pattern to yield a dot product greater than 0.

We show a schematic representation of such a weight space in Fig. 4.5. There are four pattern hyperplanes, 1, 2, 3, 4 , corresponding to patterns $\mathbf{Y}_1$, $\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, respectively, and we indicate by an arrow the half-space for each in which weight vectors give dot products greater than 0. Suppose we wanted weight values that would give positive responses for patterns $\mathbf{Y}_1$, $\mathbf{Y}_3$, and $\mathbf{Y}_4$, and a negative response for pattern $\mathbf{Y}_2$. The weight point, **V**, indicated in the figure is one such set of weight values.

The question of whether or not there exists a weight vector that gives desired responses for a given set of patterns can be given a geometric interpretation. To do so involves reversing the "polarity" of those hyperplanes corresponding to patterns for which a negative response is desired. If we do that for our example above, we get the weight space diagram shown in Fig. 4.6.

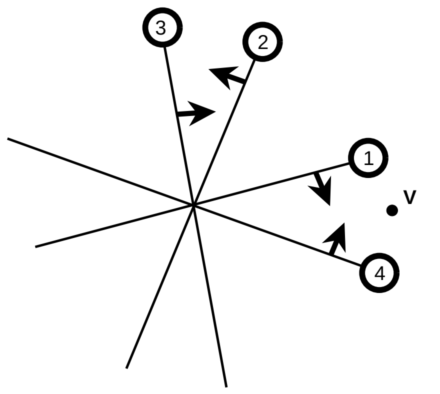


Figure 4.5: Weight Space

If a weight vector exists that correctly classifies a set of patterns, then the half-spaces defined by the correct responses for these patterns will have a non-empty intersection, called the solution region. The solution region will be a "hyper-wedge" region whose vertex is at the origin of weight space and whose cross-section increases with increasing distance from the origin. This region is shown shaded in Fig. 4.6. (The boxed numbers show, for later purposes, the number of errors made by weight vectors in each of the regions.) The fixed-increment error-correction procedure changes a weight vector by moving it normal to any pattern hyperplane for which that weight vector gives an incorrect response. Suppose in our example that we present the patterns in the sequence $\mathbf{Y}_1$, $\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, and start the process with a weight point $\mathbf{V}_1$, as shown in Fig. 4.7. Starting at $\mathbf{V}_1$, we see that it gives an incorrect response for pattern $\mathbf{Y}_1$, so we move $\mathbf{V}_1$ to $\mathbf{V}_2$ in a direction normal to plane 1. (That is what adding $\mathbf{Y}_1$ to $\mathbf{V}_1$ does.) $\mathbf{Y}_2$ gives an incorrect response for pattern $\mathbf{Y}_2$, and so on. Ultimately, the responses are only incorrect for planes bounding the solution region. Some of the subsequent corrections may overshoot the solution region, but eventually we work our way out far enough in the solution region that corrections (for a fixed increment size) take us within it. The proofs for convergence of the fixed-increment rule make this intuitive argument precise.

### 4.1.5 The Widrow-Hoff Procedure

The Widrow-Hoff procedure (also called the *LMS* or the *delta* procedure) attempts to find weights that minimize a squared-error function between the

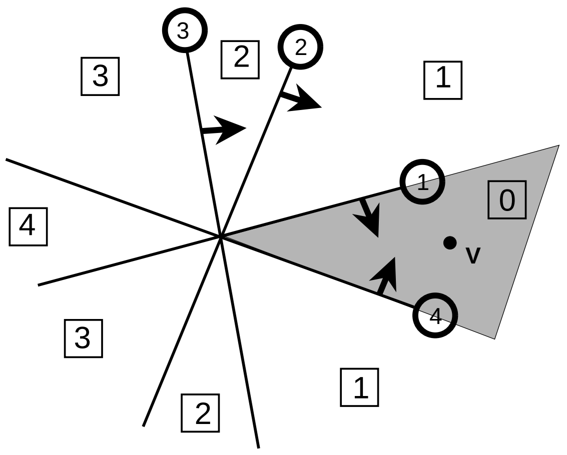


Figure 4.6: Solution Region in Weight Space

pattern labels and the dot product computed by a TLU. For this purpose, the pattern labels are assumed to be either $+1$ or $-1$ (instead of 1 or 0). The squared error for a pattern, $\mathbf{X}_i$, with label $d_i$ (for desired output) is:

$$\varepsilon_i = (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

where $x_{ij}$ is the $j$-th component of $\mathbf{X}_i$. The total squared error (over all patterns in a training set, $\Xi$, containing $m$ patterns) is then:

$$\varepsilon = \sum_{i=1}^{m} (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

We want to choose the weights $w_j$ to minimize this squared error. One way to find such a set of weights is to start with an arbitrary weight vector and move it along the negative gradient of $\varepsilon$ as a function of the weights. Since $\varepsilon$ is quadratic in the $w_j$, we know that it has a global minimum, and thus this *steepest descent* procedure is guaranteed to find the minimum. Each component of the gradient is the partial derivative of $\varepsilon$ with respect to one of the weights. One problem with taking the partial derivative of $\varepsilon$ is that $\varepsilon$ depends on *all* the input vectors in $\Xi$. Often, it is preferable to use an incremental procedure in which we try the TLU on just one element, $\mathbf{X}_i$,

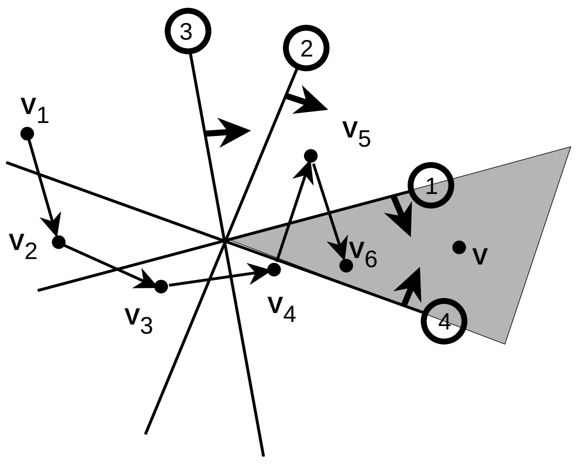


Figure 4.7: Moving Into the Solution Region

of $\Xi$ at a time, compute the gradient of the single-pattern squared error, $\varepsilon_i$, make the appropriate adjustment to the weights, and then try another member of $\Xi$. Of course, the results of the incremental version can only approximate those of the batch one, but the approximation is usually quite effective. We will be describing the incremental version here.

The $j$-th component of the gradient of the single-pattern error is:

$$\frac{\partial \varepsilon_i}{\partial w_j} = -2(d_i - \sum_{j=1}^{n+1} x_{ij} w_j) x_{ij}$$

An adjustment in the direction of the negative gradient would then change each weight as follows:

$$w_j \longleftarrow w_j + c_i(d_i - f_i)x_{ij}$$

where $f_i = \sum_{j=1}^{n+1} x_{ij} w_j$, and $c_i$ governs the size of the adjustment. The entire weight vector (in augmented, or $\mathbf{V}$, notation) is thus adjusted according to the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i(d_i - f_i)\mathbf{Y}_i$$

where, as before, $\mathbf{Y}_i$ is the $i$-th augmented pattern vector.

The Widrow-Hoff procedure makes adjustments to the weight vector whenever the dot product itself, $\mathbf{Y}_i \bullet \mathbf{V}$, does not equal the specified desired target value, $d_i$ (which is either 1 or $-1$). The learning-rate factor, $c_i$, might decrease with time toward 0 to achieve asymptotic convergence. The Widrow-Hoff formula for changing the weight vector has the same form as the standard fixed-increment error-correction formula. The only difference is that $f_i$ is the thresholded response of the TLU in the error-correction case while it is the dot product itself for the Widrow-Hoff procedure.

Finding weight values that give the desired dot products corresponds to solving a set of linear equalities, and the Widrow-Hoff procedure can be interpreted as a descent procedure that attempts to minimize the mean-squared-error between the actual and desired values of the dot product. (For more on Widrow-Hoff and other related procedures, see [Duda & Hart, 1973, pp. 151ff].)

Examples of training curves for TLU's; performance on training set; performance on test set; cumulative number of corrections.

### 4.1.6 Training a TLU on Non-Linearly-Separable Training Sets

When the training set is not linearly separable (perhaps because of noise or perhaps inherently), it may still be desired to find a "best" separating hyperplane. Typically, the error-correction procedures will not do well on non-linearly-separable training sets because they will continue to attempt to correct inevitable errors, and the hyperplane will never settle into an acceptable place.

Several methods have been proposed to deal with this case. First, we might use the Widrow-Hoff procedure, which (although it will not converge to zero error on non-linearly separable problems) will give us a weight vector that minimizes the mean-squared-error. A mean-squared-error criterion often gives unsatisfactory results, however, because it prefers many small errors to a few large ones. As an alternative, error correction with a continuous decrease toward zero of the value of the learning rate constant, $c$, will result in ever decreasing changes to the hyperplane. Duda [Duda, 1966] has suggested keeping track of the average value of the weight vector during error correction and using this average to give a separating hyperplane that performs reasonably well on non-linearly-separable problems. Gallant [Gallant, 1986] proposed what he called the "pocket algorithm." As described in [Hertz, Krogh, & Palmer, 1991, p. 160]:

> . . . the pocket algorithm . . . consists simply in storing (or "putting in your pocket") the set of weights which has had the longest unmodified run of successes so far. The algorithm is stopped after some chosen time $t$ . . .

After stopping, the weights in the pocket are used as a set that should give a small number of errors on the training set. Error-correction proceeds as usual with the ordinary set of weights.

Also see methods proposed by [John, 1995] and by [Marchand & Golea, 1993]. The latter is claimed to outperform the pocket algorithm.

## 4.2 Linear Machines

The natural generalization of a (two-category) TLU to an $R$-category classifier is the structure, shown in Fig. 4.8, called a *linear machine*. Here, to use more familiar notation, the $\mathbf{W}$s and $\mathbf{X}$ are meant to be augmented vectors (with an (n+1)-st component). Such a structure is also sometimes called a "competitive" net or a "winner-take-all" net. The output of the linear machine is one of the numbers, $\{1, \ldots, R\}$, corresponding to which dot product is largest. Note that when $R = 2$, the linear machine reduces to a TLU with weight vector $\mathbf{W} = (\mathbf{W}_1 - \mathbf{W}_2)$.

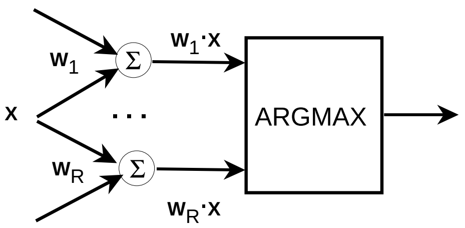


Figure 4.8: A Linear Machine

The diagram in Fig. 4.9 shows the character of the regions in a 2-dimensional space created by a linear machine for $R = 5$. In $n$ dimensions, every pair of regions is either separated by a section of a hyperplane or is non-adjacent.

To train a linear machine, there is a straightforward generalization of the 2-category error-correction rule. Assemble the patterns in the training set into a sequence as before.

1. If the machine classifies a pattern correctly, no change is made to any of the weight vectors.
2. If the machine mistakenly classifies a category $u$ pattern, $\mathbf{X}_i$, in category $v$ ($u \neq v$), then:

$$\mathbf{W}_u \longleftarrow \mathbf{W}_u + c_i \mathbf{X}_i$$

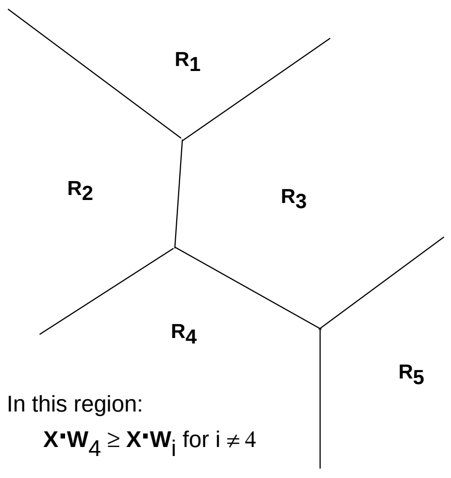


Figure 4.9: Regions For a Linear Machine

and

$$\mathbf{W}_v \longleftarrow \mathbf{W}_v - c_i \mathbf{X}_i$$

and all other weight vectors are not changed.

This correction increases the value of the $u$-th dot product and decreases the value of the $v$-th dot product. Just as in the 2-category fixed increment procedure, this procedure is guaranteed to terminate, for constant $c_i$, if there exists weight vectors that make correct separations of the training set. Note that when $R = 2$, this procedure reduces to the ordinary TLU error-correction procedure. A proof that this procedure terminates is given in [Nilsson, 1990, pp. 88-90] and in [Duda & Hart, 1973, pp. 174-177].

## 4.3 Networks of TLUs

### 4.3.1 Motivation and Examples

#### Layered Networks

To classify correctly all of the patterns in non-linearly-separable training sets requires separating surfaces more complex than hyperplanes. One way to achieve more complex surfaces is with networks of TLUs. Consider, for example, the 2-dimensional, even parity function, $f = x_1x_2 + \overline{x_1}\ \overline{x_2}$. No single line through the 2-dimensional square can separate the vertices (1,1) and (0,0) from the vertices (1,0) and (0,1)—the function is not linearly separable and thus cannot be implemented by a single TLU. But, the network of three TLUs shown in Fig. 4.10 does implement this function. In the figure, we show the weight values along input lines to each TLU and the threshold value inside the circle representing the TLU.

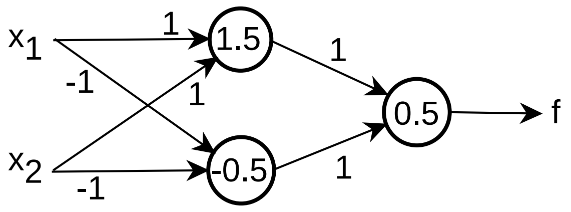


Figure 4.10: A Network for the Even Parity Function

The function implemented by a network of TLUs depends on its topology as well as on the weights of the individual TLUs. *Feedforward* networks have no cycles; in a feedforward network no TLU's input depends (through zero or more intermediate TLUs) on that TLU's output. (Networks that are not feedforward are called *recurrent* networks). If the TLUs of a feedforward network are arranged in layers, with the elements of layer $j$ receiving inputs only from TLUs in layer $j - 1$, then we say that the network is a *layered, feedforward network*. The network shown in Fig. 4.10 is a layered, feedforward network having two layers (of weights). (Some people count the layers of TLUs and include the inputs as a layer also; they would call this network a three-layer network.) In general, a feedforward, layered network has the structure shown in Fig. 4.11. All of the TLUs except the "output" units are called hidden units (they are "hidden" from the output).

### Implementing DNF Functions by Two-Layer Networks

We have already defined $k$-term DNF functions—they are DNF functions having $k$ terms. A $k$-term DNF function can be implemented by a two-layer network with $k$ units in the hidden layer—to implement the $k$ terms—and one output unit to implement the disjunction of these terms. Since any Boolean function has a DNF form, any Boolean function can be implemented by some two-layer network of TLUs. As an example, consider the

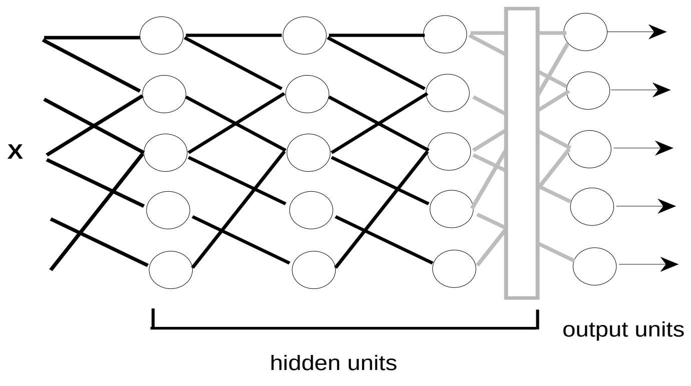


Figure 4.11: A Layered, Feedforward Network

function $f = x_1x_2 + x_2\overline{x_3} + x_1\overline{x_3}$. The form of the network that implements this function is shown in Fig. 4.12. (We leave it to the reader to calculate appropriate values of weights and thresholds.) The 3-cube representation of the function is shown in Fig. 4.13. The network of Fig. 4.12 can be designed so that each hidden unit implements one of the planar boundaries shown in Fig. 4.13.

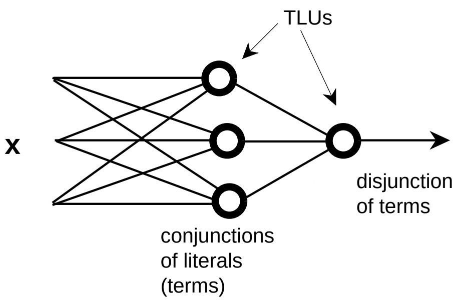


Figure 4.12: A Two-Layer Network

To train a two-layer network that implements a $k$-term DNF function, we first note that the output unit implements a disjunction, so the weights in the final layer are fixed. The weights in the first layer (except for the "threshold weights") can all have values of 1, $-1$, or 0. Later, we will

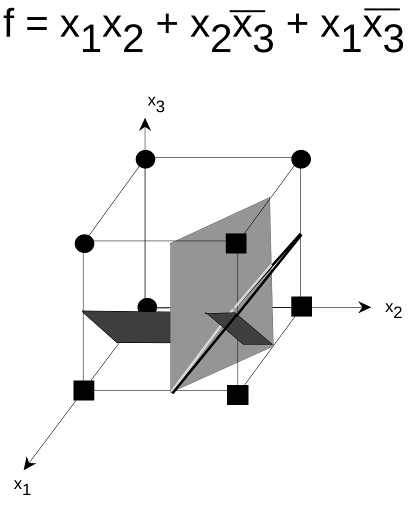


Figure 4.13: Three Planes Implemented by the Hidden Units

present a training procedure for this first layer of weights.

Discuss half-space intersections, half-space unions, NP-hardness of optimal versions, single-side-error-hypeplane methods, relation to "AQ" methods.

Add diagrams showing the non-linear transformation performed by a layered network.

### Important Comment About Layered Networks

Adding additional layers cannot compensate for an inadequate first layer of TLUs. The first layer of TLUs partitions the feature space so that no two differently labeled vectors are in the same region (that is, so that no two such vectors yield the same set of outputs of the first-layer units). If the first layer does not partition the feature space in this way, then regardless of what subsequent layers do, the final outputs will not be consistent with the labeled training set.

## 4.3.2 Madalines

### Two-Category Networks

An interesting example of a layered, feedforward network is the two-layer one which has an odd number of hidden units, and a "vote-taking" TLU as the output unit. Such a network was called a "Madaline" (for <u>m</u>any <u>adaline</u>s by Widrow. Typically, the response of the vote taking unit is defined to be the response of the majority of the hidden units, although other output logics are possible. Ridgway [Ridgway, 1962] proposed the following error-correction rule for adjusting the weights of the hidden units of a Madaline:

- If the Madaline correctly classifies a pattern, $\mathbf{X}_i$, no corrections are made to any of the hidden units' weight vectors,
- If the Madaline incorrectly classifies a pattern, $\mathbf{X}_i$, then determine the minimum number of hidden units whose responses need to be changed (from 0 to 1 or from 1 to 0—depending on the type of error) in order that the Madaline would correctly classify $\mathbf{X}_i$. Suppose that minimum number is $k_i$. Of those hidden units voting incorrectly, change the weight vectors of those $k_i$ of them whose dot products are closest to 0 by using the error correction rule:

  $$\mathbf{W} \longleftarrow \mathbf{W} + c_i(d_i - f_i)\mathbf{X}_i$$

  where $d_i$ is the desired response of the hidden unit (0 or 1) and $f_i$ is the actual response (0 or 1). (We assume augmented vectors here even though we are using $\mathbf{X}$, $\mathbf{W}$ notation.)

That is, we perform error-correction on just enough hidden units to correct the vote to a majority voting correctly, and we change those that are easiest to change. There are example problems in which even though a set of weight values exists for a given Madaline structure such that it could classify all members of a training set correctly, this procedure will fail to find them. Nevertheless, the procedure works effectively in most experiments with it.

We leave it to the reader to think about how this training procedure could be modified if the output TLU implemented an *or* function (or an *and* function).

### $R$-Category Madalines and Error-Correcting Output Codes

If there are $k$ hidden units ($k > 1$) in a two-layer network, their responses correspond to vertices of a $k$-dimensional hypercube. The ordinary two-category Madaline identifies two special points in this space, namely the vertex consisting of $k$ 1's and the vertex consisting of $k$ 0's. The Madaline's response is 1 if the point in "hidden-unit-space" is closer to the all 1's vertex than it is to the all 0's vertex. We could design an $R$-category Madaline by identifying $R$ vertices in hidden-unit space and then classifying a pattern according to which of these vertices the hidden-unit response is closest to. A machine using that idea was implemented in the early 1960s at SRI [Brain, *et al.*, 1962]. It used the fact that the $2^p$ so-called *maximal-length shift-register sequences* [Peterson, 1961, pp. 147ff] in a $(2^p - 1)$-dimensional Boolean space are mutually equidistant (for any integer $p$). For similar, more recent work see [Dietterich & Bakiri, 1991].

### 4.3.3 Piecewise Linear Machines

A two-category training set is linearly separable if there exists a threshold function that correctly classifies all members of the training set. Similarly, we can say that an $R$-category training set is linearly separable if there exists a linear machine that correctly classifies all members of the training set. When an $R$-category problem is not linearly separable, we need a more powerful classifier. A candidate is a structure called a *piecewise linear (PWL)* machine illustrated in Fig. 4.14.

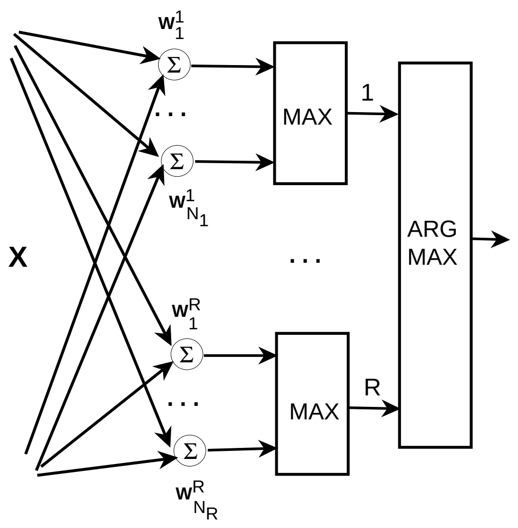


Figure 4.14: A Piecewise Linear Machine

The PWL machine groups its weighted summing units into $R$ banks corresponding to the $R$ categories. An input vector $\mathbf{X}$ is assigned to that category corresponding to the bank with the largest weighted sum. We can use an error-correction training algorithm similar to that used for a linear machine. If a pattern is classified incorrectly, we subtract (a constant times) the pattern vector from the weight vector producing the largest dot product (it was incorrectly the largest) and add (a constant times) the pattern vector to that weight vector in the correct bank of weight vectors whose dot product is locally largest in that bank. (Again, we use augmented vectors here.) Unfortunately, there are example training sets that are separable by a given PWL machine structure but for which this error-correction training method fails to find a solution. The method does appear to work well in some situations [Duda & Fossum, 1966], although [Nilsson, 1965, page 89] observed that "it is probably not a very effective method for training PWL machines having more than three [weight vectors] in each bank."

### 4.3.4 Cascade Networks

Another interesting class of feedforward networks is that in which all of the TLUs are ordered and each TLU receives inputs from all of the pattern components and from all TLUs lower in the ordering. Such a network is called a *cascade* network. An example is shown in Fig. 4.15 in which the TLUs are labeled by the linearly separable functions (of their inputs) that they implement. Each TLU in the network implements a set of $2^k$ parallel hyperplanes, where $k$ is the number of TLUs from which it receives inputs. (Each of the $k$ preceding TLUs can have an output of 1 or 0; that's $2^k$ different combinations—resulting in $2^k$ different positions for the parallel hyperplanes.) We show a 3-dimensional sketch for a network of two TLUs in Fig. 4.16. The reader might consider how the $n$-dimensional parity function might be implemented by a cascade network having $\log_2 n$ TLUs.

Cascade networks might be trained by first training $L_1$ to do as good a job as possible at separating all the training patterns (perhaps by using the pocket algorithm, for example), then training $L_2$ (including the weight from $L_1$ to $L_2$) also to do as good a job as possible at separating all the training patterns, and so on until the resulting network classifies the patterns in the training set satisfactorily.

Also mention the "cascade-correlation" method of [Fahlman & Lebiere, 1990].

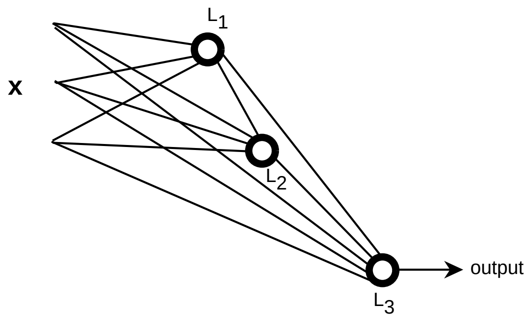


Figure 4.15: A Cascade Network

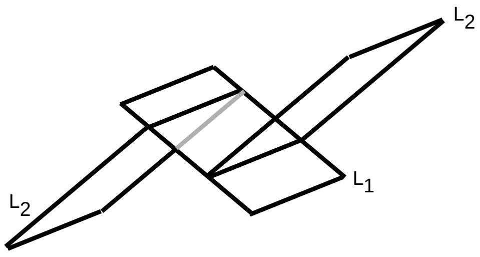


Figure 4.16: Planes Implemented by a Cascade Network with Two TLUs

## 4.4 Training Feedforward Networks by Backpropagation

### 4.4.1 Notation

The general problem of training a network of TLUs is difficult. Consider, for example, the layered, feedforward network of Fig. 4.11. If such a network makes an error on a pattern, there are usually several different ways in which the error can be corrected. It is difficult to assign "blame" for the error to any particular TLU in the network. Intuitively, one looks for weight-adjusting procedures that move the network in the correct direction (relative to the error) by making minimal changes. In this spirit, the Widrow-Hoff method of gradient descent has been generalized to deal with multilayer networks.

In explaining this generalization, we use Fig. 4.17 to introduce some notation. This network has only one output unit, but, of course, it is possible to have several TLUs in the output layer—each implementing a different function. Each of the layers of TLUs will have outputs that we take to be the components of vectors, just as the input features are components of an input vector. The $j$-th layer of TLUs ($1 \leq j < k$) will have as their outputs the vector $\mathbf{X}^{(j)}$. The input feature vector is denoted by $\mathbf{X}^{(0)}$, and the final output (of the $k$-th layer TLU) is $f$. Each TLU in each layer has a weight vector (connecting it to its inputs) and a threshold; the $i$-th TLU in the $j$-th layer has a weight vector denoted by $\mathbf{W}_i^{(j)}$. (We will assume that the "threshold weight" is the last component of the associated weight vector; we might have used $\mathbf{V}$ notation instead to include this threshold component, but we have chosen here to use the familiar $\mathbf{X}$,$\mathbf{W}$ notation, assuming that these vectors are "augmented" as appropriate.) We denote the weighted sum input to the $i$-th threshold unit in the $j$-th layer by $s_i^{(j)}$. (That is, $s_i^{(j)} = \mathbf{X}^{(j-1)} \bullet \mathbf{W}_i^{(j)}$.) The number of TLUs in the $j$-th layer is given by $m_j$. The vector $\mathbf{W}_i^{(j)}$ has components $w_{l,i}^{(j)}$ for $l = 1, m_{(j-1)} + 1$.

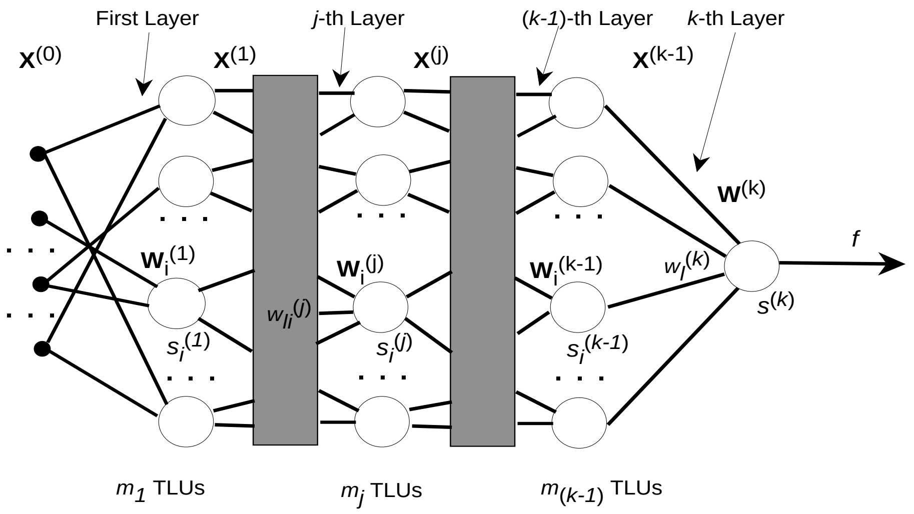


Figure 4.17: A $k$-layer Network

### 4.4.2 The Backpropagation Method

A gradient descent method, similar to that used in the Widrow Hoff method, has been proposed by various authors for training a multi-layer, feedforward network. As before, we define an error function on the final output of the network and we adjust each weight in the network so as to minimize the error. If we have a desired response, $d_i$, for the $i$-th input vector, $\mathbf{X}_i$, in the training set, $\Xi$, we can compute the squared error over the entire training set to be:

$$\varepsilon = \sum_{\mathbf{X}_i \; \epsilon \; \Xi} (d_i - f_i)^2$$

where $f_i$ is the actual response of the network for input $\mathbf{X}_i$. To do gradient descent on this squared error, we adjust each weight in the network by an amount proportional to the negative of the partial derivative of $\varepsilon$ with respect to that weight. Again, we use a single-pattern error function so that we can use an incremental weight adjustment procedure. The squared error for a single input vector, $\mathbf{X}$, evoking an output of $f$ when the desired output is $d$ is:

$$\varepsilon = (d - f)^2$$

It is convenient to take the partial derivatives of $\varepsilon$ with respect to the various weights in groups corresponding to the weight vectors. We define a partial derivative of a quantity $\phi$, say, with respect to a weight vector, $\mathbf{W}_i^{(j)}$, thus:

$$\frac{\partial \phi}{\partial \mathbf{W}_i^{(j)}} \overset{\text{def}}{=} \left[ \frac{\partial \phi}{\partial w_{1i}^{(j)}}, \ldots, \frac{\partial \phi}{\partial w_{li}^{(j)}}, \ldots, \frac{\partial \phi}{\partial w_{m_{j-1}+1,i}^{(j)}} \right]$$

where $w_{li}^{(j)}$ is the $l$-th component of $\mathbf{W}_i^{(j)}$. This vector partial derivative of $\phi$ is called the *gradient* of $\phi$ with respect to $\mathbf{W}$ and is sometimes denoted by $\nabla_{\mathbf{W}}\phi$.

Since $\varepsilon$'s dependence on $\mathbf{W}_i^{(j)}$ is entirely through $s_i^{(j)}$, we can use the chain rule to write:

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = \frac{\partial \varepsilon}{\partial s_i^{(j)}} \frac{\partial s_i^{(j)}}{\partial \mathbf{W}_i^{(j)}}$$

Because $s_i^{(j)} = \mathbf{X}^{(j-1)} \bullet \mathbf{W}_i^{(j)}$, $\frac{\partial s_i^{(j)}}{\partial \mathbf{W}_i^{(j)}} = \mathbf{X}^{(j-1)}$. Substituting yields:

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = \frac{\partial \varepsilon}{\partial s_i^{(j)}} \mathbf{X}^{(j-1)}$$

Note that $\frac{\partial \varepsilon}{\partial s_i^{(j)}} = -2(d-f)\frac{\partial f}{\partial s_i^{(j)}}$. Thus,

$$\frac{\partial \varepsilon}{\partial \mathbf{W}_i^{(j)}} = -2(d-f)\frac{\partial f}{\partial s_i^{(j)}} \mathbf{X}^{(j-1)}$$

The quantity $(d-f)\frac{\partial f}{\partial s_i^{(j)}}$ plays an important role in our calculations; we shall denote it by $\delta_i^{(j)}$. Each of the $\delta_i^{(j)}$'s tells us how sensitive the squared error of the network output is to changes in the input to each threshold function. Since we will be changing weight vectors in directions along their negative gradient, our fundamental rule for weight changes throughout the network will be:

$$\mathbf{W}_i^{(j)} \leftarrow \mathbf{W}_i^{(j)} + c_i^{(j)} \delta_i^{(j)} \mathbf{X}^{(j-1)}$$

where $c_i^{(j)}$ is the learning rate constant for this weight vector. (Usually, the learning rate constants for all weight vectors in the network are the same.) We see that this rule is quite similar to that used in the error correction procedure for a single TLU. A weight vector is changed by the addition of a constant times its vector of (unweighted) inputs.

Now, we must turn our attention to the calculation of the $\delta_i^{(j)}$'s. Using the definition, we have:

$$\delta_i^{(j)} = (d-f)\frac{\partial f}{\partial s_i^{(j)}}$$

We have a problem, however, in attempting to carry out the partial derivatives of $f$ with respect to the $s$'s. The network output, $f$, is not continuously differentiable with respect to the $s$'s because of the presence of the threshold functions. Most small changes in these sums do not change $f$ at all, and when $f$ does change, it changes abruptly from 1 to 0 or vice versa.

A way around this difficulty was proposed by Werbos [Werbos, 1974] and (perhaps independently) pursued by several other researchers, for example [Rumelhart, Hinton, & Williams, 1986]. The trick involves replacing all the threshold functions by differentiable functions called *sigmoids*.[1] The output of a sigmoid function, superimposed on that of a threshold function, is shown in Fig. 4.18. Usually, the sigmoid function used is $f(s) = \frac{1}{1+e^{-s}}$ , where $s$ is the input and $f$ is the output.

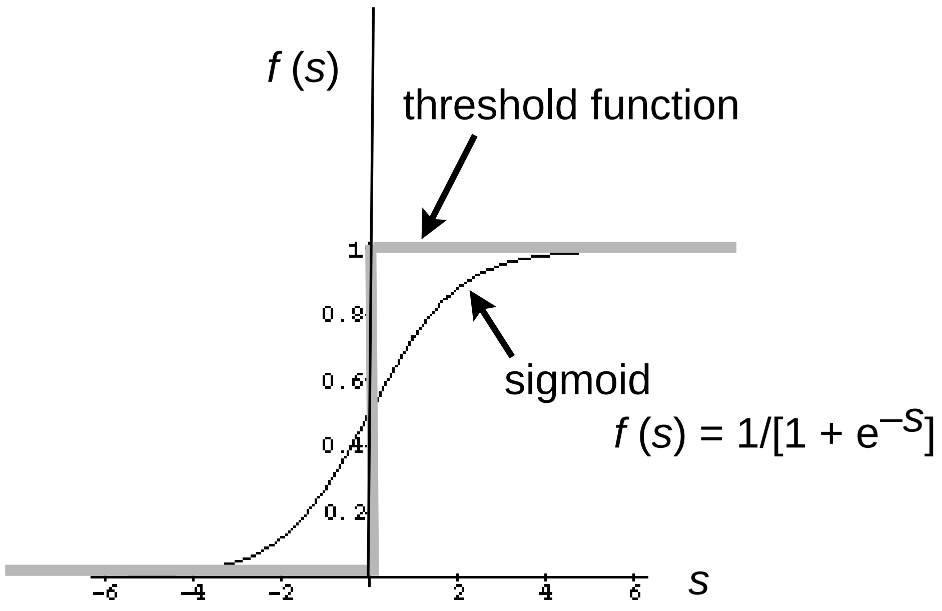


Figure 4.18: A Sigmoid Function

We show the network containing sigmoid units in place of TLUs in Fig. 4.19. The output of the $i$-th sigmoid unit in the $j$-th layer is denoted by $f_i^{(j)}$. (That is, $f_i^{(j)} = \frac{1}{1+e^{-s_i^{(j)}}}$.)

### 4.4.3 Computing Weight Changes in the Final Layer

We first calculate $\delta^{(k)}$ in order to compute the weight change for the final sigmoid unit:

[1][Russell & Norvig 1995, page 595] attributes the use of this idea to [Bryson & Ho 1969].

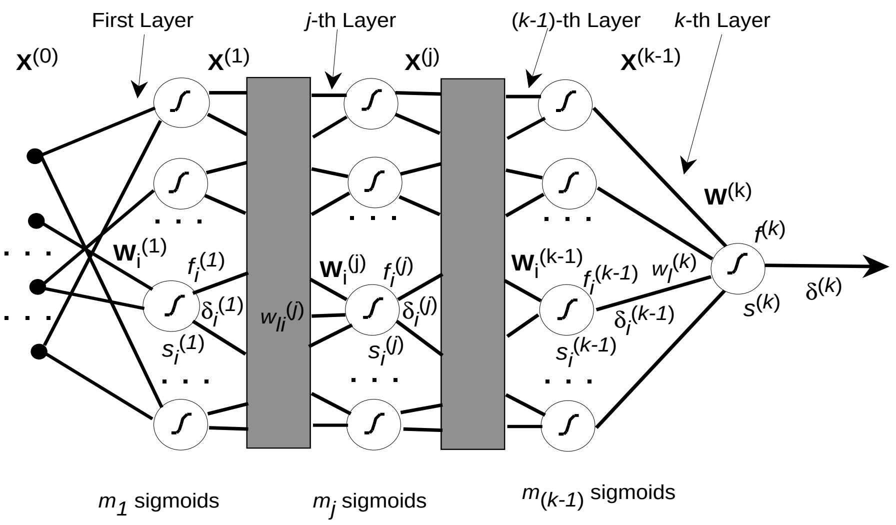


Figure 4.19: A Network with Sigmoid Units

$$\delta^{(k)} = (d - f^{(k)})\frac{\partial f^{(k)}}{\partial s^{(k)}}$$

Given the sigmoid function that we are using, namely $f(s) = \frac{1}{1+e^{-s}}$, we have that $\frac{\partial f}{\partial s} = f(1-f)$. Substituting gives us:

$$\delta^{(k)} = (d - f^{(k)})f^{(k)}(1 - f^{(k)})$$

Rewriting our general rule for weight vector changes, the weight vector in the final layer is changed according to the rule:

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} + c^{(k)}\delta^{(k)}\mathbf{X}^{(k-1)}$$

where $\delta^{(k)} = (d - f^{(k)})f^{(k)}(1 - f^{(k)})$

It is interesting to compare backpropagation to the error-correction rule and to the Widrow-Hoff rule. The backpropagation weight adjustment for the single element in the final layer can be written as:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)f(1 - f)\mathbf{X}$$

Written in the same format, the error-correction rule is:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)\mathbf{X}$$

and the Widrow-Hoff rule is:

$$\mathbf{W} \longleftarrow \mathbf{W} + c(d - f)\mathbf{X}$$

The only difference (except for the fact that $f$ is not thresholded in Widrow-Hoff) is the $f(1 - f)$ term due to the presence of the sigmoid function. With the sigmoid function, $f(1 - f)$ can vary in value from 0 to 1. When $f$ is 0, $f(1 - f)$ is also 0; when $f$ is 1, $f(1 - f)$ is 0; $f(1 - f)$ obtains its maximum value of 1/4 when $f$ is 1/2 (that is, when the input to the sigmoid is 0). The sigmoid function can be thought of as implementing a "fuzzy" hyperplane. For a pattern far away from this fuzzy hyperplane, $f(1 - f)$ has value close to 0, and the backpropagation rule makes little or no change to the weight values regardless of the desired output. (Small changes in the weights will have little effect on the output for inputs far from the hyperplane.) Weight changes are only made within the region of "fuzz" surrounding the hyperplane, and these changes are in the direction of correcting the error, just as in the error-correction and Widrow-Hoff rules.

### 4.4.4 Computing Changes to the Weights in Intermediate Layers

Using our expression for the $\delta$'s, we can similarly compute how to change each of the weight vectors in the network. Recall:

$$\delta_i^{(j)} = (d - f)\frac{\partial f}{\partial s_i^{(j)}}$$

Again we use a chain rule. The final output, $f$, depends on $s_i^{(j)}$ through each of the summed inputs to the sigmoids in the $(j + 1)$-th layer. So:

$$\delta_i^{(j)} = (d - f)\frac{\partial f}{\partial s_i^{(j)}}$$

$$= (d - f)\left[\frac{\partial f}{\partial s_1^{(j+1)}}\frac{\partial s_1^{(j+1)}}{\partial s_i^{(j)}} + \cdots + \frac{\partial f}{\partial s_l^{(j+1)}}\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} + \cdots + \frac{\partial f}{\partial s_{m_{j+1}}^{(j+1)}}\frac{\partial s_{m_{j+1}}^{(j+1)}}{\partial s_i^{(j)}}\right]$$

$$= \sum_{l=1}^{m_{j+1}} (d-f)\frac{\partial f}{\partial s_l^{(j+1)}}\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)}\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}}$$

It remains to compute the $\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}}$'s. To do that we first write:

$$s_l^{(j+1)} = \mathbf{X}^{(j)}\bullet\mathbf{W}_l^{(j+1)}$$

$$= \sum_{\nu=1}^{m_j+1} f_\nu^{(j)} w_{\nu l}^{(j+1)}$$

And then, since the weights do not depend on the $s$'s:

$$\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = \frac{\partial\left[\sum_{\nu=1}^{m_j+1} f_\nu^{(j)} w_{\nu l}^{(j+1)}\right]}{\partial s_i^{(j)}} = \sum_{\nu=1}^{m_j+1} w_{\nu l}^{(j+1)}\frac{\partial f_\nu^{(j)}}{\partial s_i^{(j)}}$$

Now, we note that $\frac{\partial f_\nu^{(j)}}{\partial s_i^{(j)}} = 0$ unless $\nu = i$, in which case $\frac{\partial f_\nu^{(j)}}{\partial s_\nu^{(j)}} = f_\nu^{(j)}(1 - f_\nu^{(j)})$. Therefore:

$$\frac{\partial s_l^{(j+1)}}{\partial s_i^{(j)}} = w_{il}^{(j+1)} f_i^{(j)}(1 - f_i^{(j)})$$

We use this result in our expression for $\delta_i^{(j)}$ to give:

$$\delta_i^{(j)} = f_i^{(j)}(1 - f_i^{(j)}) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{il}^{(j+1)}$$

The above equation is recursive in the $\delta$'s. (It is interesting to note that this expression is independent of the error function; the error function explicitly affects only the computation of $\delta^{(k)}$.) Having computed the $\delta_i^{(j+1)}$'s for layer $j+1$, we can use this equation to compute the $\delta_i^{(j)}$'s. The base case is $\delta^{(k)}$, which we have already computed:

$$\delta^{(k)} = (d - f^{(k)})f^{(k)}(1 - f^{(k)})$$

We use this expression for the $\delta$'s in our generic weight changing rule, namely:

$$\mathbf{W}_i^{(j)} \leftarrow \mathbf{W}_i^{(j)} + c_i^{(j)}\delta_i^{(j)}\mathbf{X}^{(j-1)}$$

Although this rule appears complex, it can be simply implemented by "backpropagating" the $\delta$'s through the weights in order to adjust the amount by which $\mathbf{X}$ vectors are added to or subtracted from weight vectors (thus, the name *backprop* for this algorithm).

Add additional intuitive explanation.

### 4.4.5 Variations on Backprop

[To be written: problem of local minima, simulated annealing, momemtum (Plaut, *et al.*, 1986, see [Hertz, Krogh, & Palmer, 1991]), quickprop, regularization methods]

#### Simulated Annealing

To apply simulated annealing, the value of the learning rate constant is gradually decreased with time. If we fall early into an error-function valley that is not very deep (a local minimum), it typically will neither be very broad, and soon a subsequent large correction will jostle us out of it. It is less likely that we will move out of deep valleys, and at the end of the process (with very small values of the learning rate constant), we descend to its deepest point. The process gets its name by analogy with annealing in metallurgy, in which a material's temperature is gradually decreased allowing its crystalline structure to reach a minimal energy state.

### 4.4.6 An Application: Steering a Van

A neural network system called ALVINN (Autonomous Land Vehicle in a Neural Network) has been trained to steer a Chevy van successfully on ordinary roads and highways at speeds of 55 mph [Pomerleau, 1991, Pomerleau, 1993]. The input to the network is derived from a low-resolution (30 x 32) television image. The TV camera is mounted on the van and looks at the road straight ahead. This image is sampled and produces a stream of 960-dimensional input vectors to the neural network. The network is shown in Fig. 4.20.

The network has five hidden units in its first layer and 30 output units in the second layer; all are sigmoid units. The output units are arranged in

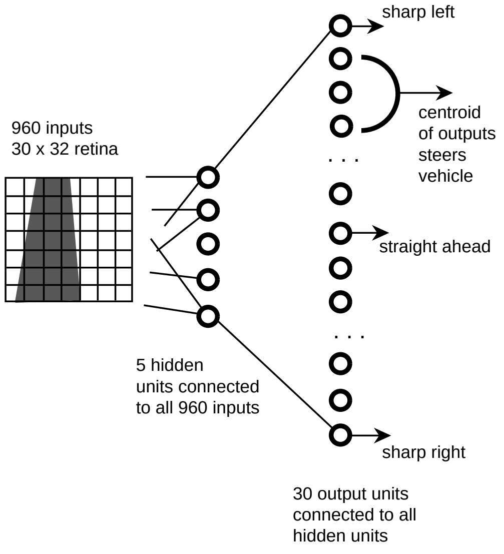

sharp left
centroid of outputs steers vehicle
960 inputs
30 x 32 retina
straight ahead
5 hidden units connected to all 960 inputs
sharp right
30 output units connected to all hidden units

Figure 4.20: The ALVINN Network

a linear order and control the van's steering angle. If a unit near the top of the array of output units has a higher output than most of the other units, the van is steered to the left; if a unit near the bottom of the array has a high output, the van is steered to the right. The "centroid" of the responses of all of the output units is computed, and the van's steering angle is set at a corresponding value between hard left and hard right.

The system is trained by a modified on-line training regime. A driver drives the van, and his actual steering angles are taken as the correct labels for the corresponding inputs. The network is trained incrementally by backprop to produce the driver-specified steering angles in response to each visual pattern as it occurs in real time while driving.

This simple procedure has been augmented to avoid two potential problems. First, since the driver is usually driving well, the network would never get any experience with far-from-center vehicle positions and/or incorrect vehicle orientations. Also, on long, straight stretches of road, the network would be trained for a long time only to produce straight-ahead steering angles; this training would swamp out earlier training to follow a curved road. We wouldn't want to try to avoid these problems by instructing the driver to drive erratically occasionally, because the system would learn to mimic this erratic behavior.

Instead, each original image is shifted and rotated in software to create 14 additional images in which the vehicle appears to be situated differently relative to the road. Using a model that tells the system what steering angle ought to be used for each of these shifted images, given the driver-specified steering angle for the original image, the system constructs an additional 14 labeled training patterns to add to those encountered during ordinary driver training.

## 4.5 Synergies Between Neural Network and Knowledge-Based Methods

To be written; discuss rule-generating procedures (such as [Towell & Shavlik, 1992]) and how expert-provided rules can aid neural net training and vice-versa [Towell, Shavlik, & Noordweier, 1990].

## 4.6 Bibliographical and Historical Remarks

To be added.