Machine Learning

Chapter 04

Artificial Neural Networks

Artificial Neural Networks

Introduction

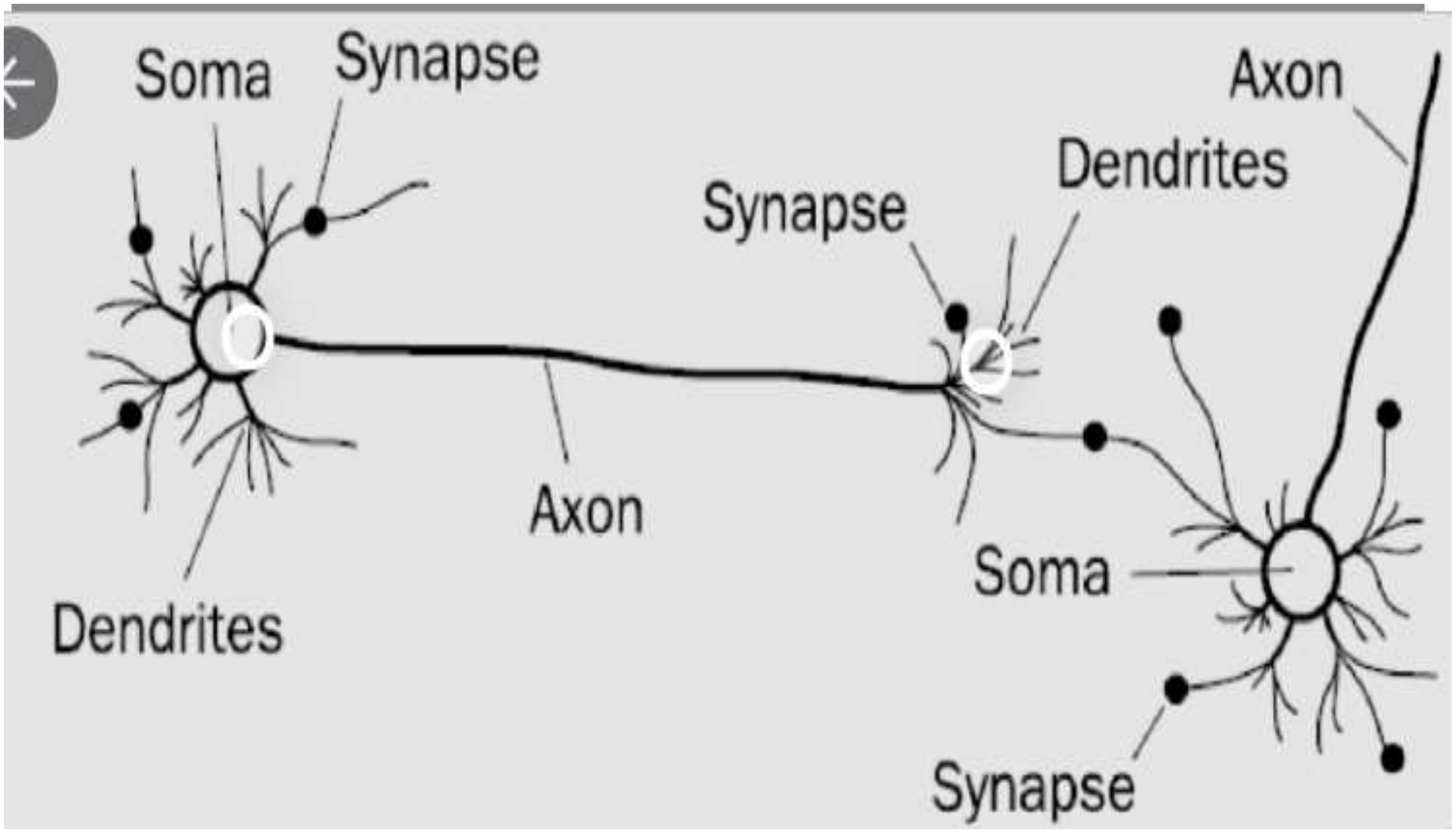
- Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.
- learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies
- learning to recognize handwritten characters, spoken words, and to recognize faces.
- For problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Biological Motivation

- Artificial neural networks are built out of a densely interconnected set of simple units
- where each unit takes a number of real-valued inputs, possibly the outputs of other units and produces a single real-valued output.
- which may become the input to many other units.

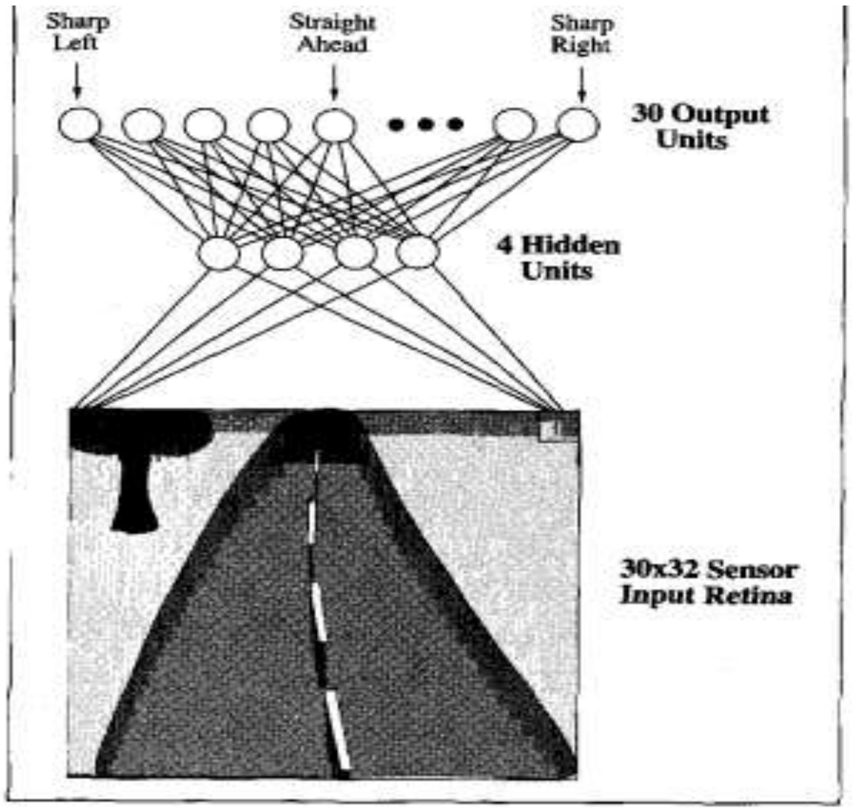
- The human brain, is estimated to contain a densely interconnected network of approximately 10^11 neurons, each connected, on average, to 10^4 others.
- Neuron activity is typically excited or inhibited through connections to other neurons.
- The fastest neuron switching times are known to be on the order of 10^-3 seconds-quite slow compared to computer switching speeds of 10^-10 seconds.
- Yet humans are able to make surprisingly complex decisions, surprisingly quickly.

Biological Neuron



NEURAL NETWORK REPRESENTATIONS

- An example of ANN learning is provided by system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways.
- The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered.
- ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways



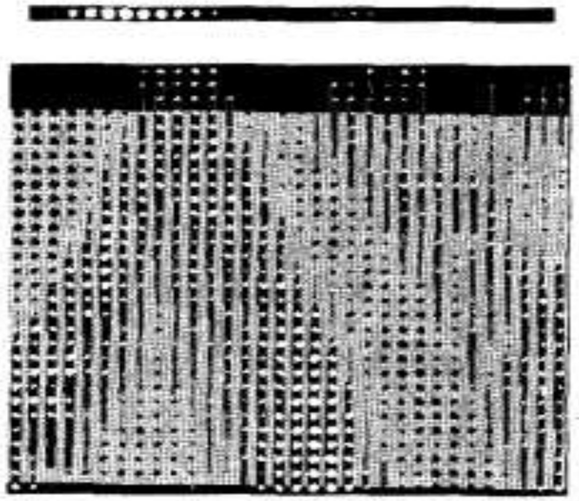
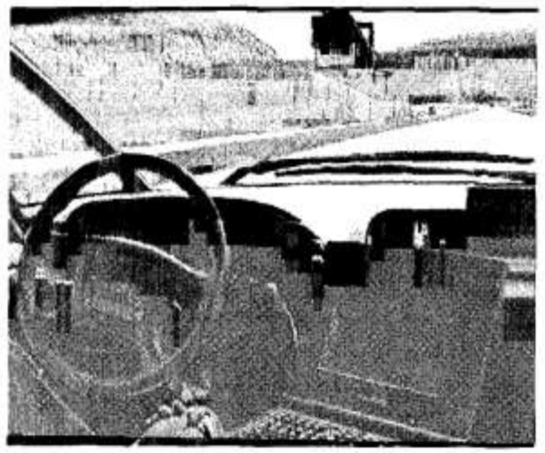


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

- Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system,
- The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.
- there are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs.

- These hidden unit outputs are then used

as inputs to a second layer of 30 "output“ units. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.



- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit.
- Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

- ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.
- It is appropriate for problems with the following characteristics:

- **Instances are represented by many attribute-value pairs:-**The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.

- **The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes:-** For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.

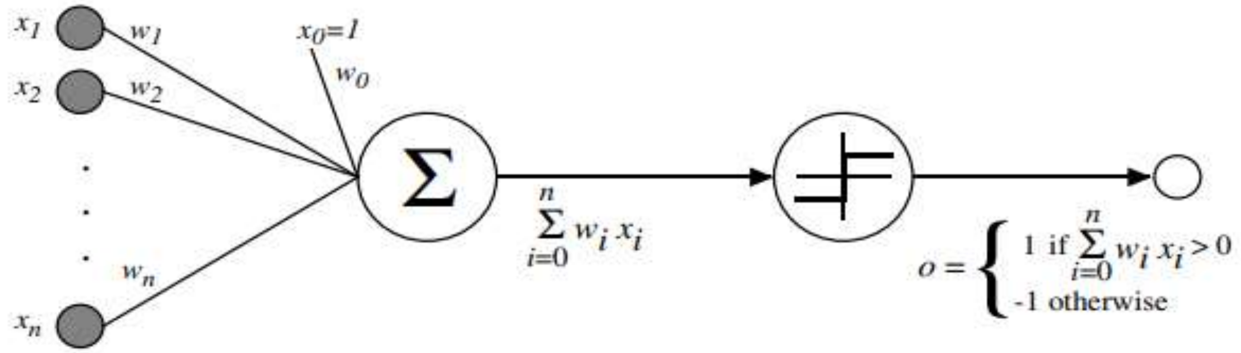
- **The training examples may contain errors**:- ANN learning methods are quite robust to noise in the training data.

- **Long training times are acceptable:-** Network training algorithms typically require longer training times than, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered
- **Fast evaluation of the learned target function may be required**:- ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.

- **The ability of humans to understand the learned target function is not important**:-The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

PERCEPTRONS

- ANN system is based on a unit called a perceptron.
- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs x1 through xn, the output o(x1, . . . , xn) computed by the perceptron



x_1
w_1
x_2
w_2
$x_0=1$
w_0
w_n
x_n
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- Notice the quantity ($-w_O$) is a threshold that the weighted combination of inputs $w_l x_l + \ldots + w_n x_n$ must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

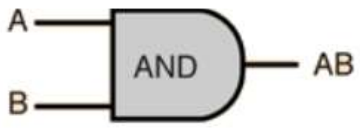
where

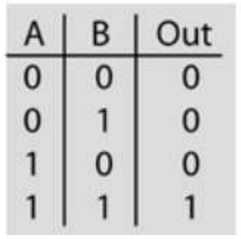
$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

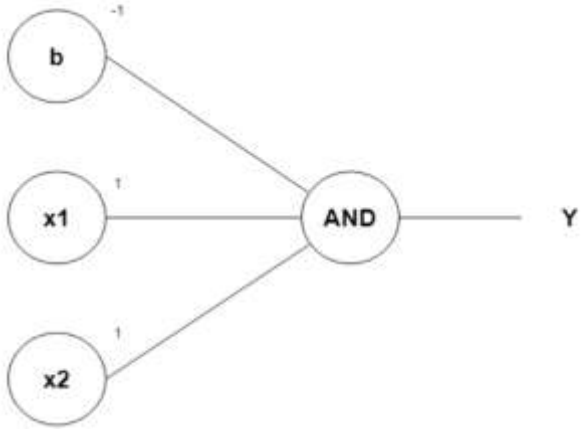
$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

AND operation





A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1



Row 1

•From w1*x1 + w2*x2 + b, initializing w1, w2, as 1 and b as –1, we get;

x1(1)+x2(1)–1

•Passing the first row of the AND logic table (x1=0, x2=0), we get;

0+0–1 = –1

•From the Perceptron rule, if Wx+b ≤ 0, then y`=0. Therefore, this row is correct, and no need for Backpropagation.

Row 2

•Passing (x1=0 and x2=1), we get;

0+1–1 = 0

•From the Perceptron rule, if Wx+b≤0, then y`=0. This row is correct, as the output is 0 for the AND gate.

•From the Perceptron rule, this works (for both row 1, row 2 and 3).

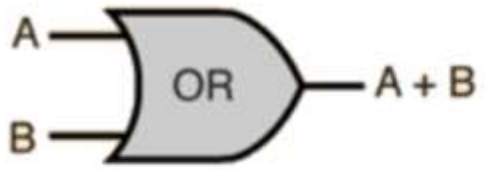
Row 4

•Passing (x1=1 and x2=1), we get;

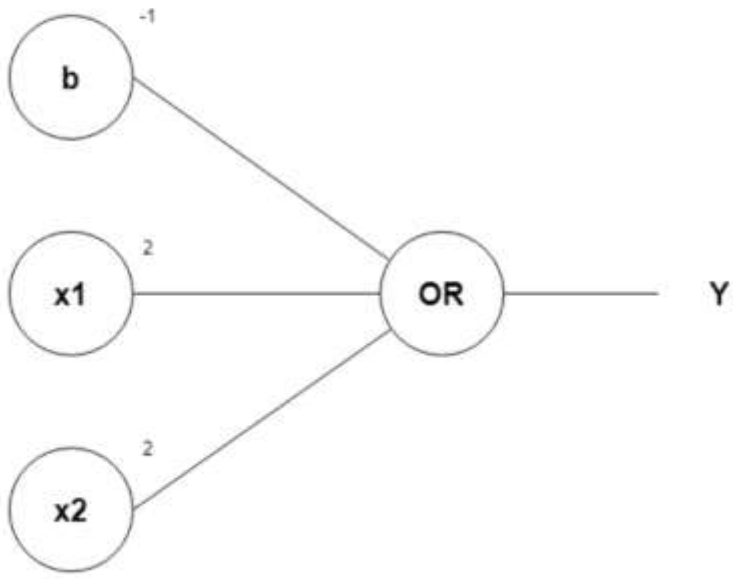
1+1–1 = 1

•Again, from the perceptron rule, this is still valid.

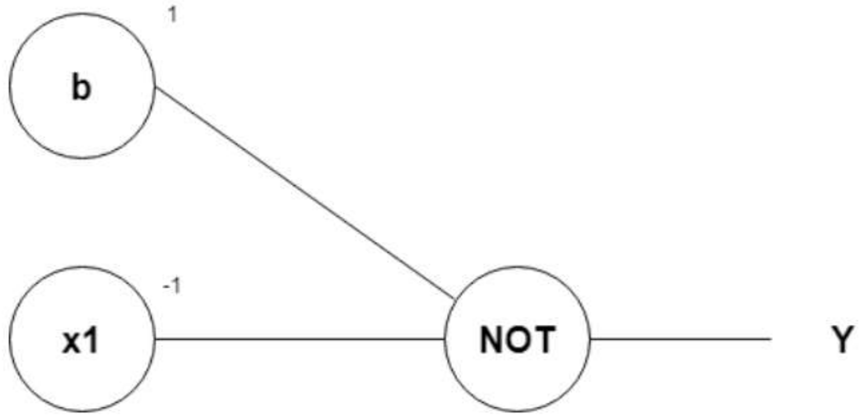
OR operation

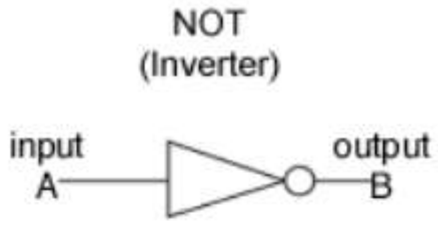


A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1



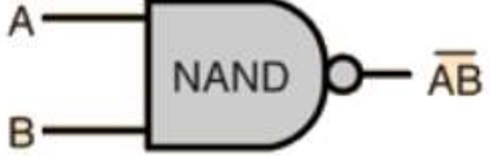
NOT operation



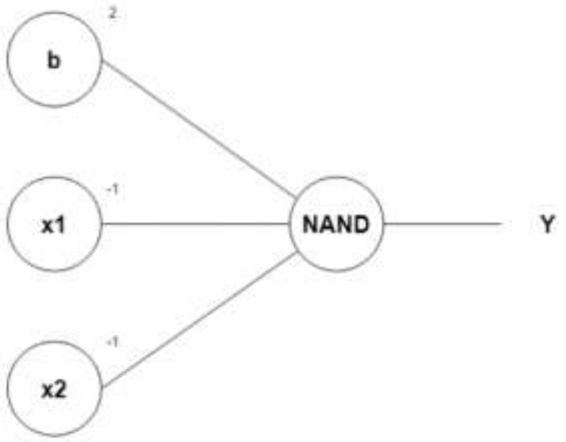


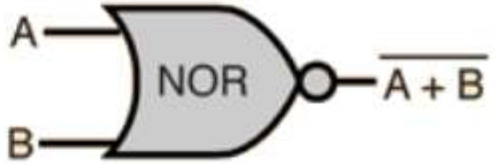
A	B
0	1
1	0

NOT Gate

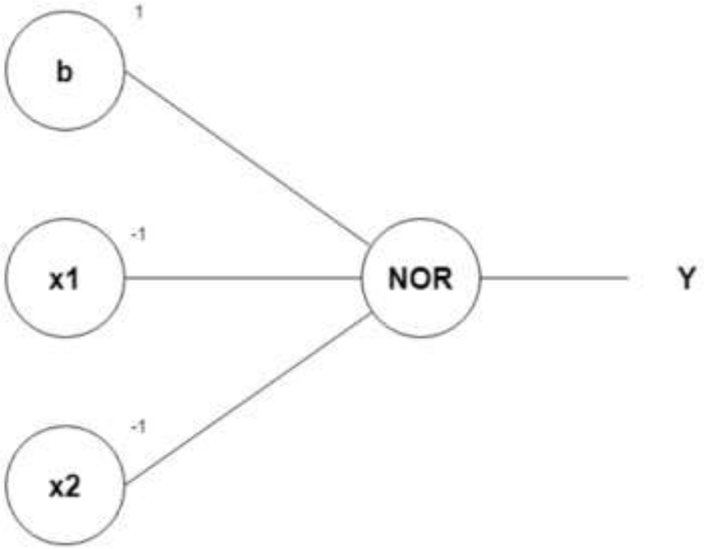


A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0





A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0



Representational Power of Perceptrons

- We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points).
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w}.\vec{x}$=0
- some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.
- A single perceptron can be used to represent many boolean functions.
- For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights w0 = -0.8, and w1 = w2 =0 .5. This perceptron can be made to represent the OR function instead by altering the threshold to w0 = -.3.
- Perceptrons can represent all of the primitive boolean functions AND, OR, NAND (¬ AND), and NOR (¬ OR).
- Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if x1 ≠ x2.

Decision Surface of a Perceptron

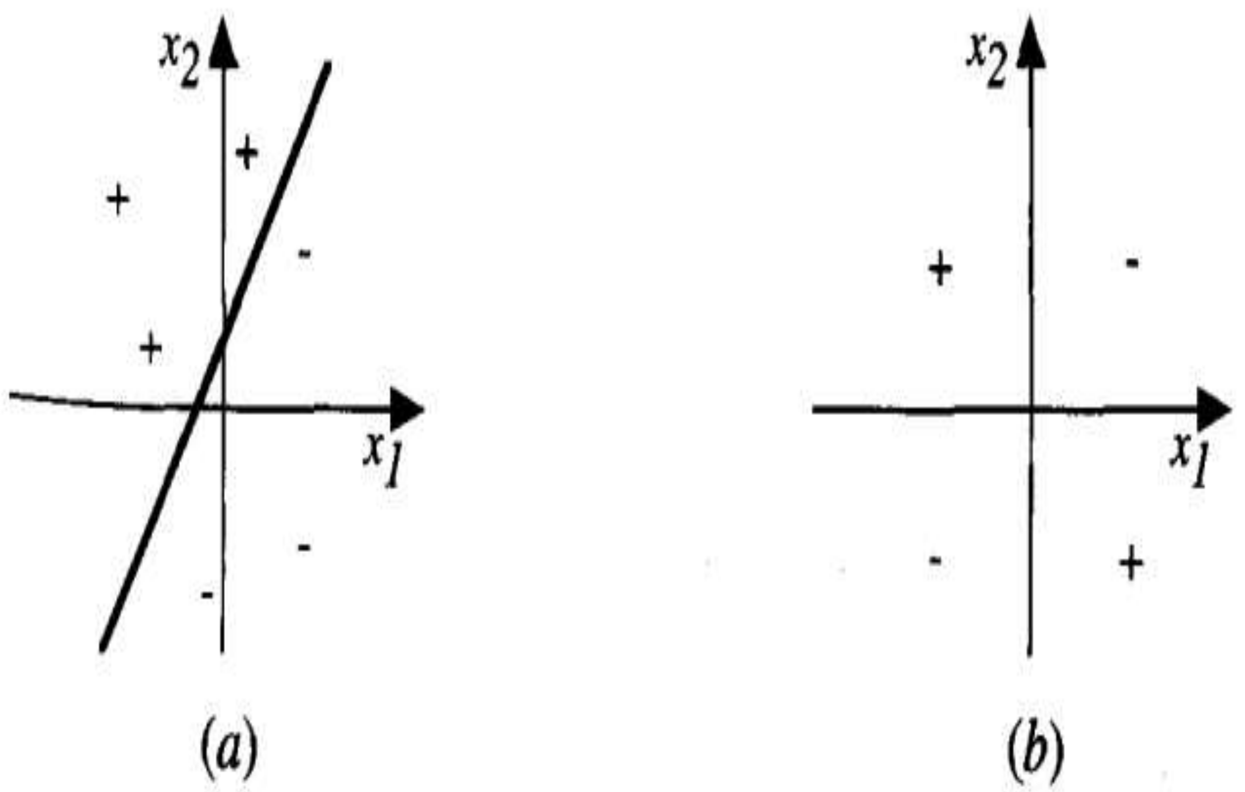


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the perceptron inputs. Positive examples are indicated by "+", negative by "–".

- Every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.
- networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

The Perceptron Training Rule

- Several algorithms are known to solve this learning problem. Here we consider two algorithms: **the perceptron rule and the delta rule** (a variant of the LMS rule for learning evaluation functions).
- These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions.
- They are important to ANNs because they provide the basis for learning networks of many units
- One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron rule to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.

- Weights are modified at each step according to the perceptron training rule, which revises the weight wi associated with input xi according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

- The role of the learning rate is to moderate the degree to which weights are changed at each step.
- It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Example

- For example, if xi = .8, n = 0.1, t = 1, and o = - 1, then the weight update will be Δwi = n(t - o)xi = O.1(1 - (-1))0.8 = 0.16.
- On the other hand, if t = -1 and o = 1, then weights associated with positive xi will be decreased rather than increased.
- In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, provided the training examples are linearly separable and provided a sufficiently small n is used.
- If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

- The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.
- A second training rule, **called the delta rule**, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The **key idea behind the delta rule is to use gradient descent to search the hypothesis space** of possible weight vectors to find the weights that best fit the training examples.
- This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units.

- The delta training rule is best understood by considering the **task of training an unthresholded perceptron; that is, a linear unit for which the output o is given by**

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

- where D is the set of training examples, t_d is the target output for training example d, and o_d is the output of the linear unit for training example d.
- By this definition, $E(\vec{w})$ is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

VISUALIZING THE HYPOTHESIS SPACE

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values, as illustrated in Figure 4.4.
- Here the axes w0 and w1 represent possible values for the two weights of a simple linear unit.
- The w0, w1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space. **we desire a hypothesis with minimum error.**
- Given the way in which we chose to define E, **for linear units this error surface must always be parabolic with a single global minimum.**
- The specific parabola will depend, of course, on the particular set of training examples.

VISUALIZING THE HYPOTHESIS SPACE

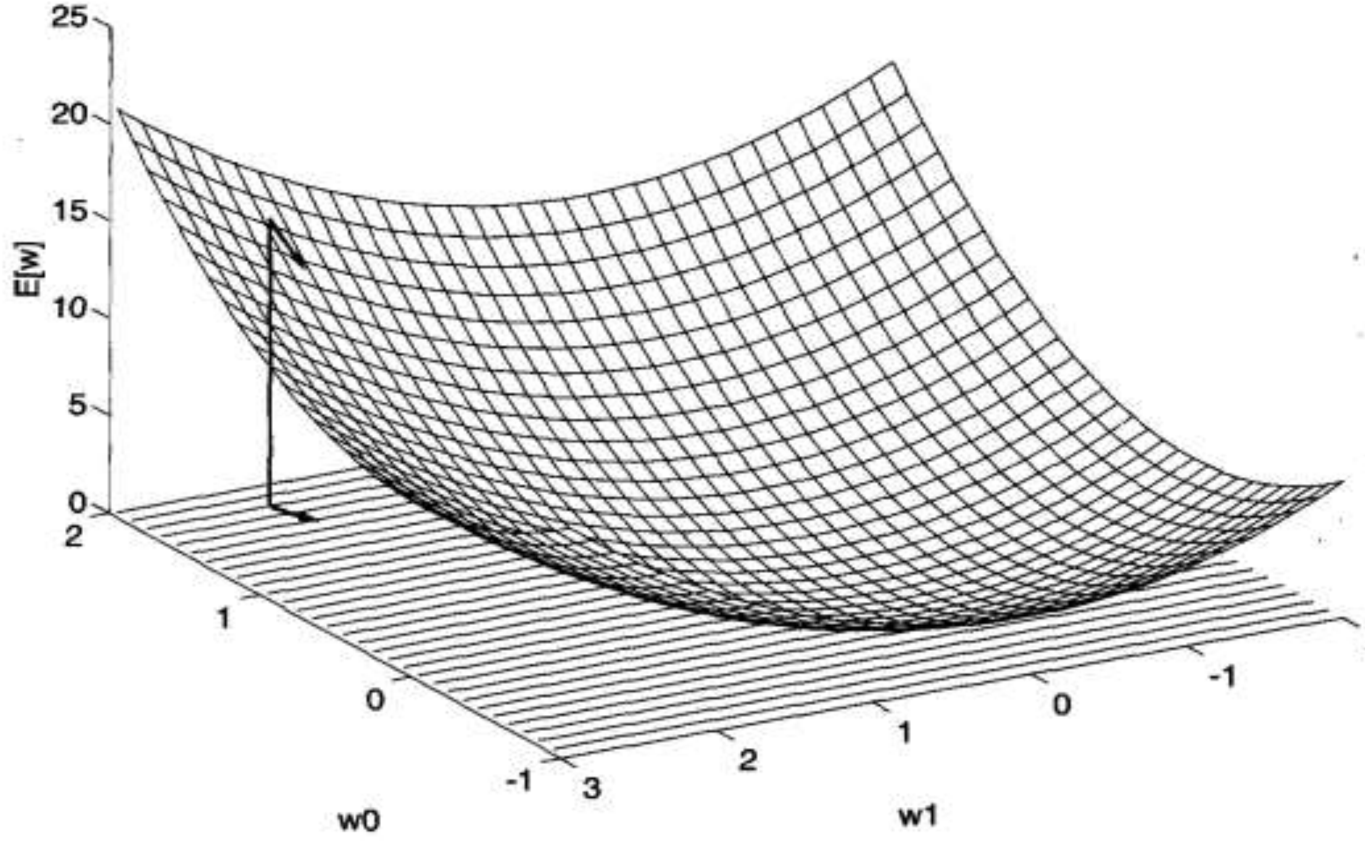
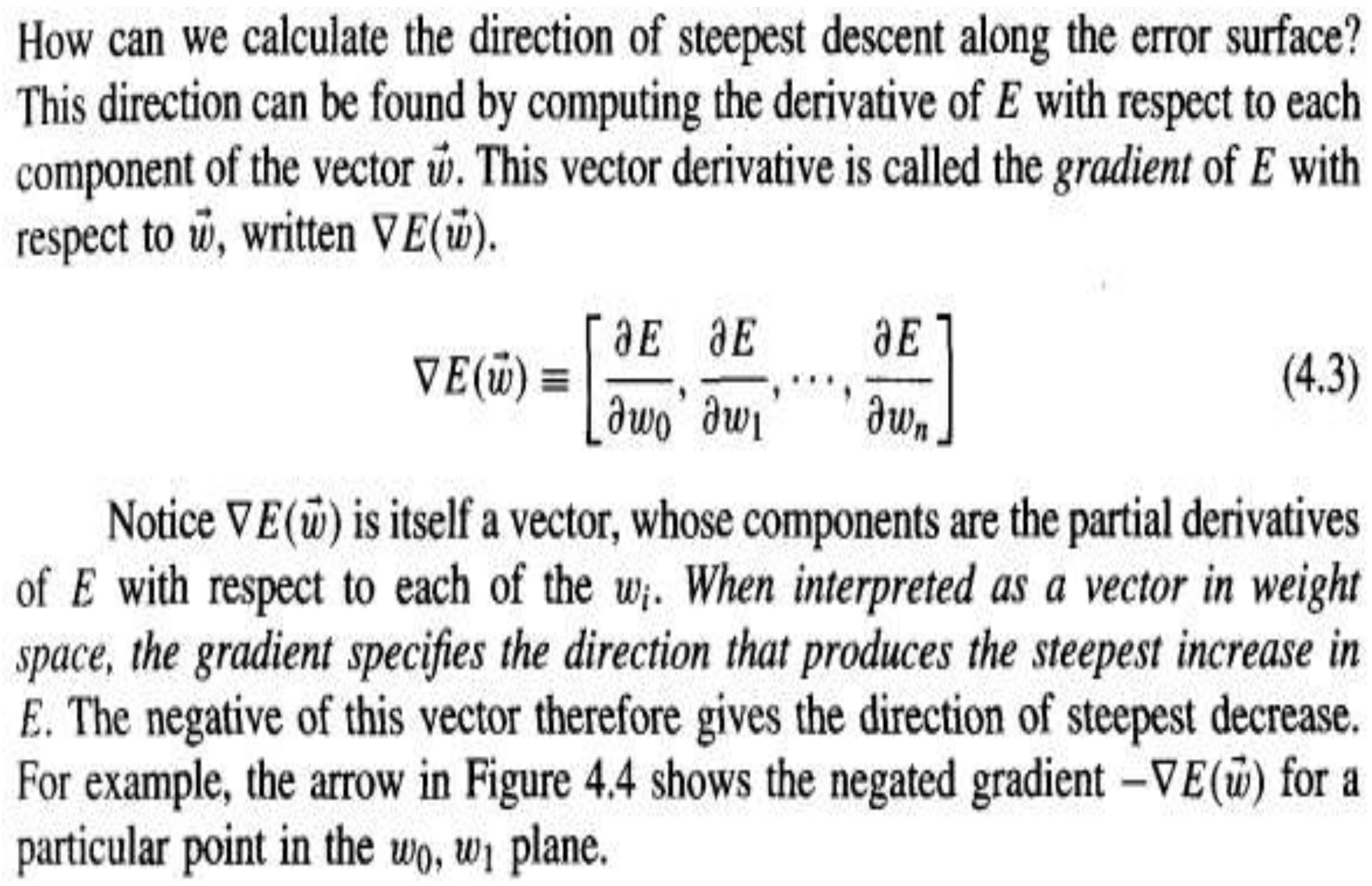


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

DERIVATION OF THE GRADIENT DESCENT RULE



How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the *gradient* of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. *When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in* E. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

•Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta \nabla E(\vec{w}) \qquad (4.4)$$

•Here n is a positive constant called the learning rate, which determines the step size in the gradient descent search.
•The negative sign is present because we want to move the weight vector in the direction that decreases E.
•This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad (4.5)$$

- To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step.
- The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from Equation (4.2)
- $$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad \text{----4.2}$$

  Where D is set of training examples

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \qquad \text{-----4.6}$$

- where x_{id} denotes the single input component x_i for training example d.
- We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs x_{id} , outputs o_d , and target values t_d associated with the training examples.
- Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \qquad \text{------ } 4.7$$

```
GRADIENT-DESCENT(training_examples, $\eta$)
    Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and
    $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
• Initialize each $w_i$ to some small random value
• Until the termination condition is met, Do
    • Initialize each $\Delta w_i$ to zero.
    • For each $\langle \vec{x}, t \rangle$ in training_examples, Do
        • Input the instance $\vec{x}$ to the unit and compute the output $o$
        • For each linear unit weight $w_i$, Do
                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$        (T4.1)
    • For each linear unit weight $w_i$, Do
                $w_i \leftarrow w_i + \Delta w_i$        (T4.2)
```

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

- To summarize, the **gradient descent algorithm** for training linear units is as follows: Pick an initial random weight vector.
- Apply the linear unit to all training examples, then
- compute Δw_i for each weight according to Equation (4.7).
- Update each weight Wi by adding Δw_i then repeat this process.
- This algorithm is given in Table 4.1. Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate n is used.
- If n is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it.
- For this reason, one common modification to the algorithm is to gradually reduce the value of n as the number of gradient descent steps grows.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

- Gradient descent is an important general paradigm for learning.
- It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever
- (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and

  (2) the error can be differentiated with respect to these hypothesis parameters.
- The key practical difficulties in applying gradient descent are

  (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and

  (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

- One common variation on gradient descent intended to alleviate these difficulties is called **incremental gradient descent, or alternatively stochastic gradient descent.**
- Whereas the gradient descent training rule presented in Equation (4.7) computes weight updates after summing over all the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example.
- The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\, x_i \tag{4.10}$$

- where t, o, and xi are the target value, unit output, and ith input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation (T4.2) is simply deleted and Equation (T4.1) replaced by

$$w_i \leftarrow w_i + \eta(t - o)\, x_i.$$

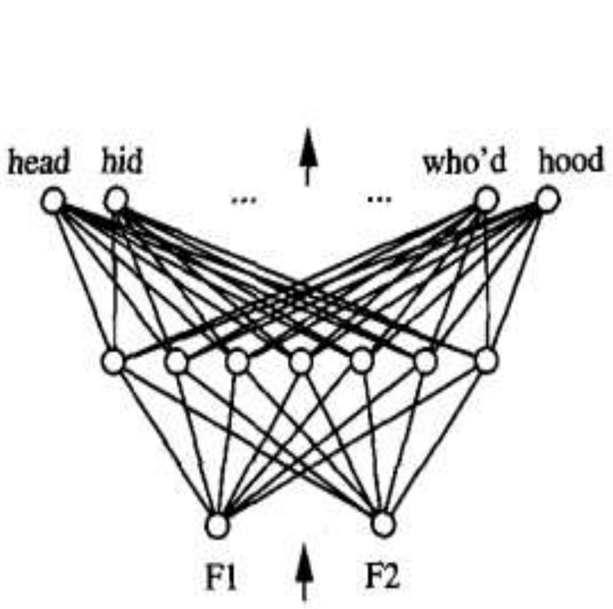
- One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

- where td, and o_d are the target value and the unit output value for training example d.
- Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$.
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function E($\overrightarrow{w}$).
- By making the value of η (the gradient descent step size) sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely
- The training rule in Equation (4.10) is known as the **delta rule**

Multi Layer Networks and The Backpropagatio Algorithm

- Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPACATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.
- For example, a typical multilayer network and decision surface is depicted in Figure 4.5.
- Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.).
- The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space.
- As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure 4.3.



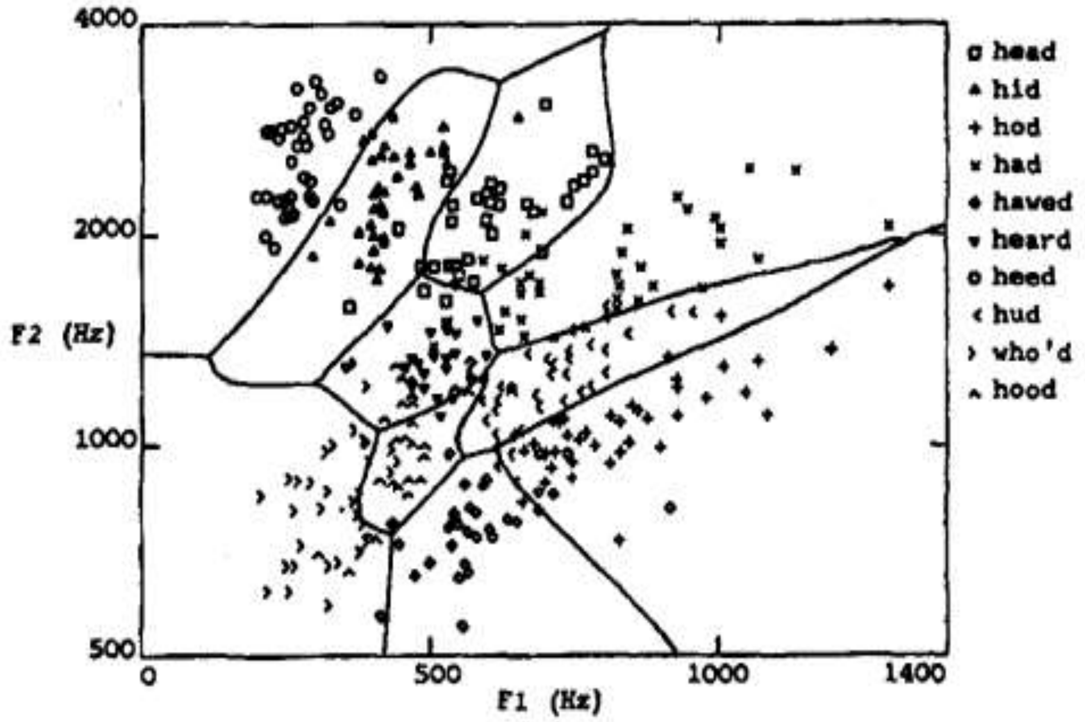


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

A Differentiable Threshold Unit

- What type of unit shall we use as the basis for constructing multilayer networks?
- The perceptron unit is a possible choice, but its discontinuous threshold makes it un differentiable and hence unsuitable for gradient descent
- What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

- The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result.
- In the case of the sigmoid unit, however, the threshold output is a continuous function of its input.

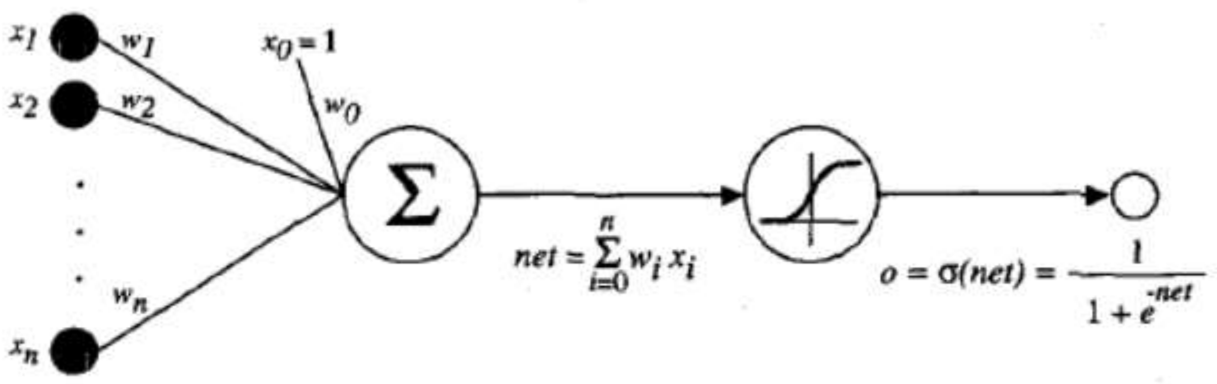

x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

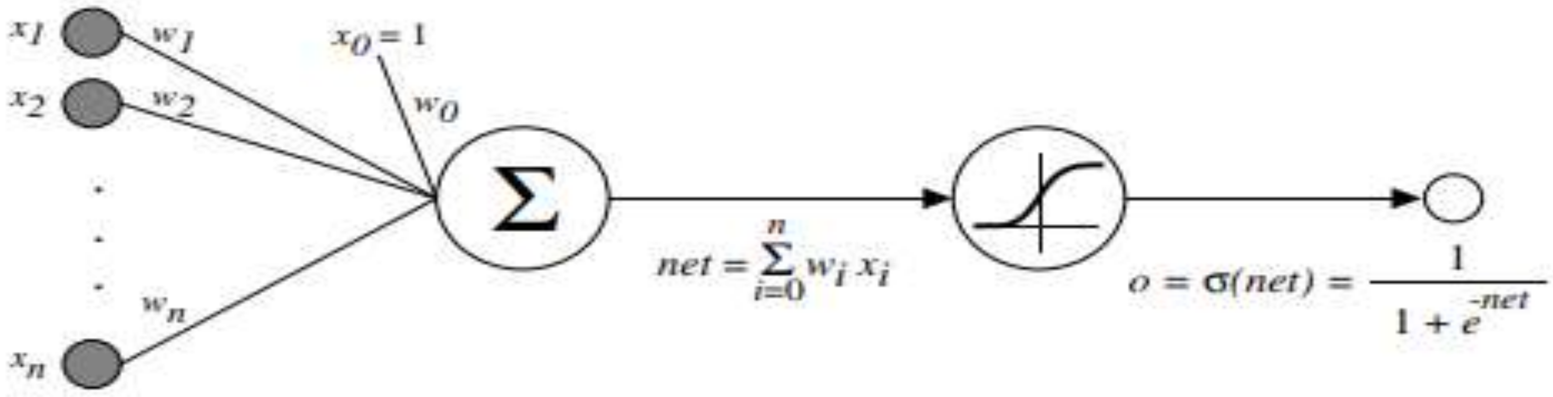
FIGURE 4.6
The sigmoid threshold unit.

- More precisely, the sigmoid unit computes its output ***o*** as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

- σ is often called the sigmoid function or, alternatively, the logistic function.
- its output ranges between 0 and 1, increasing monotonically with its input
- Because it maps a very large input domain to a small range of outputs, it is often referred to as the ***squashingfunction*** of the unit. The sigmoid function has the useful property that its derivative is easily expressed in terms of its output



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation

Error Gradient for a Sigmoid Unit

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(- \frac{\partial o_d}{\partial w_i} \right) \\
&= - \sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

BACKPROPAGATION Algorithm

- The BACKPROPAGATION algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

- where outputs is the set of output units in the network, and t_{kd} and O_{kd} are the target and output values associated with the kth output unit and training example d.

```
BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)
        Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input
        values, and $\vec{t}$ is the vector of target network output values.
        $\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of
        units in the hidden layer, and $n_{out}$ the number of output units.
        The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted
        $w_{ji}$.
• Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
• Initialize all network weights to small random numbers (e.g., between −.05 and .05).
• Until the termination condition is met, Do
    • For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do
        Propagate the input forward through the network:
        1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in
           the network.
        Propagate the errors backward through the network:
        2. For each network output unit $k$, calculate its error term $\delta_k$
                $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$                    (T4.3)
        3. For each hidden unit $h$, calculate its error term $\delta_h$
                $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$          (T4.4)
        4. Update each network weight $w_{ji}$
                $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
           where
                $\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$                          (T4.5)
```

- One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4.
- Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error.
- Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications
- The algorithm as described here applies to layered feed forward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer.
- This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION.
- The notation used here is the same as that used in earlier sections, with the following extensions:
  - An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
  - **xji denotes the input from node i to unit j,** and wji denotes the corresponding weight.
  - δ_n , denotes the error term associated with unit n. It plays a role analogous to the quantity (t - o) in our earlier discussion of the delta training rule. As we shall see later $\delta_n = -\frac{\partial E}{\partial net_n}$.

- The algorithm begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values.
- Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples.
- For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network.
- This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.
- The gradient descent weight-update rule is similar to the delta training rule. Like the delta rule, it updates each weight in proportion to the learning rate n, the input value xji to which the weight is applied, and the error in the output of the unit.
- The only difference is that the error (t - o) in the delta rule is replaced by a more complex error term, δ_j. The exact form of δ_j follows from the derivation of the weight tuning rule.
- To understand it intuitively, first consider how δ_j. is computed for each network output unit k.
- δ_j. is simply the familiar (tk - ok) from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function.
- The δ_h value for each hidden unit h . However, since training examples provide target values tk only for network outputs, no target values are directly available to indicate the error of hidden units' values.
- Instead, the error term for hidden unit h is calculated by summing the error δ_k terms for each output unit influenced by h, weighting each of the δ_k by Wkh, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

ADDING MOMENTUM

- Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. most common is to alter the weight-update rule in the algorithm by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

$$\Delta w_{ji}(n) = \eta\, \delta_j\, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- **0 < *a* < 1 *is a constant called the momentum.***
- The second term on the right is new and is called the momentum term. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of *a! is to* add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence

LEARNING IN ARBITRARY ACYCLIC NETWORKS

- The algorithm given there easily generalizes to feedforward networks of arbitrary depth.
- Only change is to the procedure for computing δ values. In general, the δ_r *value for a unit* ***r in layer m is computed from the*** δ ***values at the*** next deeper layer ***m + 1 according to***

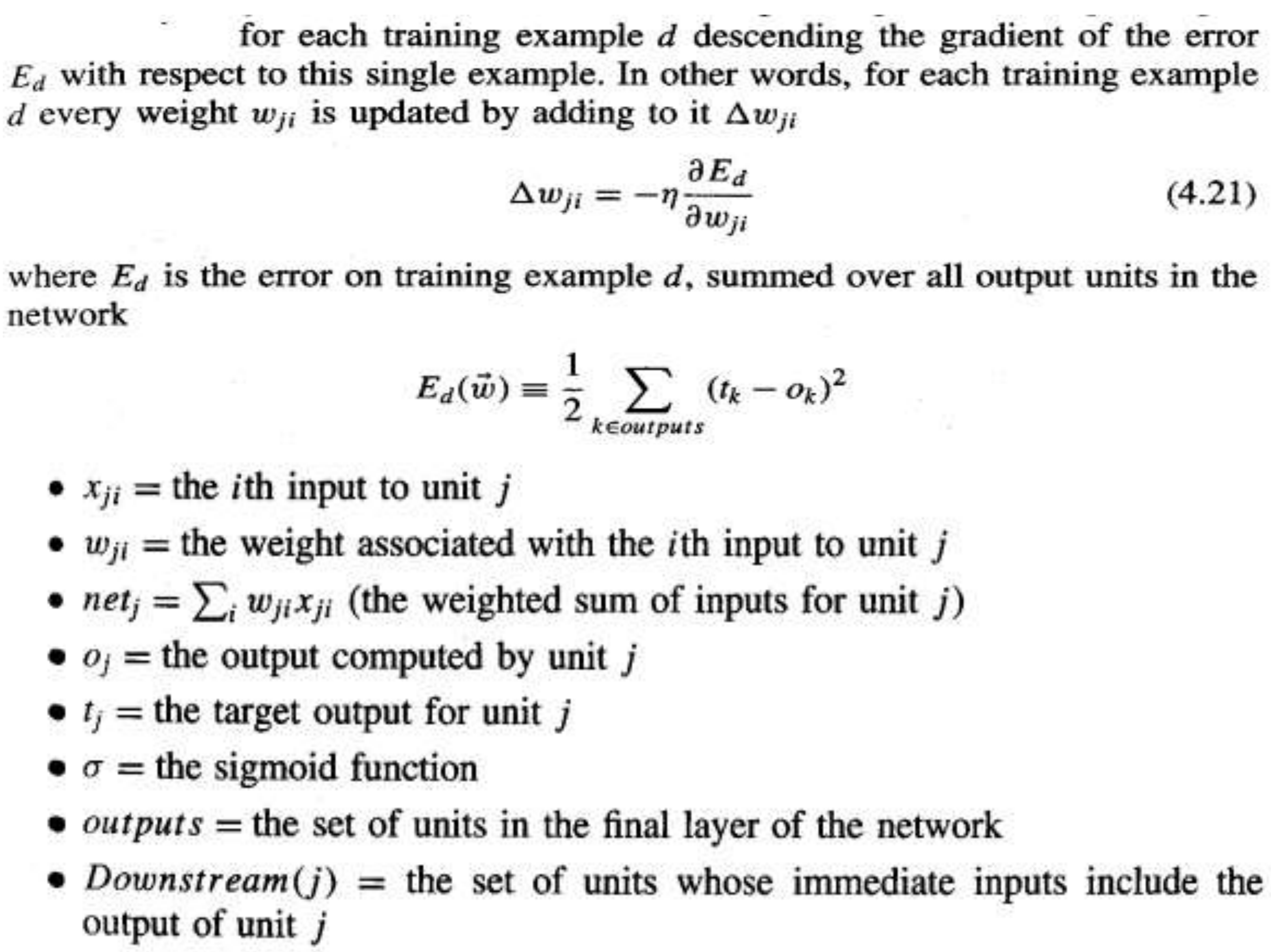
$$\delta_r = o_r\,(1 - o_r) \sum_{s \in layer\ m+1} w_{sr}\,\delta_s$$

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating δ *for any internal unit (i.e., any unit that is not an output) is*

$$\delta_r = o_r\,(1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\,\delta_s$$

where *Downstream(r) is the set of units immediately downstream from unit r in* the network: that is, all units whose inputs include the output of unit r.

Derivation of the BACKPROPAGATION Rule



for each training example d descending the gradient of the error E_d with respect to this single example. In other words, for each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{4.21}$$

where E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

- We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule .
- To begin, notice that weight w_{ji} can influence the rest of the network only through net_j. Therefore, we can use the chain rule to write

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji}\end{aligned}$$

- Given Equation (4.22), our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$.
- We consider two cases in turn: the case where unit j is an output unit for the network, and the case where j is an internal unit.

Case 1: Training Rule for Output Unit Weights. Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{4.23}$$

To begin, consider just the first term in Equation (4.23)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial (t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \qquad (4.24)\end{aligned}$$

Next consider the second term in Equation (4.23). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j) \qquad (4.25)\end{aligned}$$

Substituting expressions (4.24) and (4.25) into (4.23), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j) \qquad (4.26)$$

- and combining this with Equations (4.21) and (4.22), we have the stochastic gradient descent rule for output units.

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta \ (t_j - o_j) \ o_j(1 - o_j)x_{ji}$$

- **Case 2: Training Rule for Hidden Unit Weights.** In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for wji must take into account the indirect ways in which wji can influence the network outputs and hence Ed.
- We denote this set of units by Downstream(j). Notice that netj can influence the network outputs (and therefore Ed) only through the units in Downstream(j). Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1 - o_j)
\end{aligned} \tag{4.28}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \; w_{kj}$$

and

$$\Delta w_{ji} = \eta \; \delta_j \; x_{ji}$$

Convergence

- The BACKPROPAGATION algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs.
- Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Consider that networks with large numbers of weights correspond to error surfaces in very high dimensional spaces (one dimension per weight).
- When gradient descent falls into a local minimum with respect to one of these weights, it will not necessarily be in a local minimum with respect to the other weights.
- In fact, the more weights in the network, the more dimensions that might provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight.

- A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.
- if network weights are initialized to values near zero, then during early gradient descent steps the network will represent a very smooth function that is approximately linear in its inputs.
- This is because the sigmoid threshold function itself is approximately linear when the weights are close to zero
- Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions.

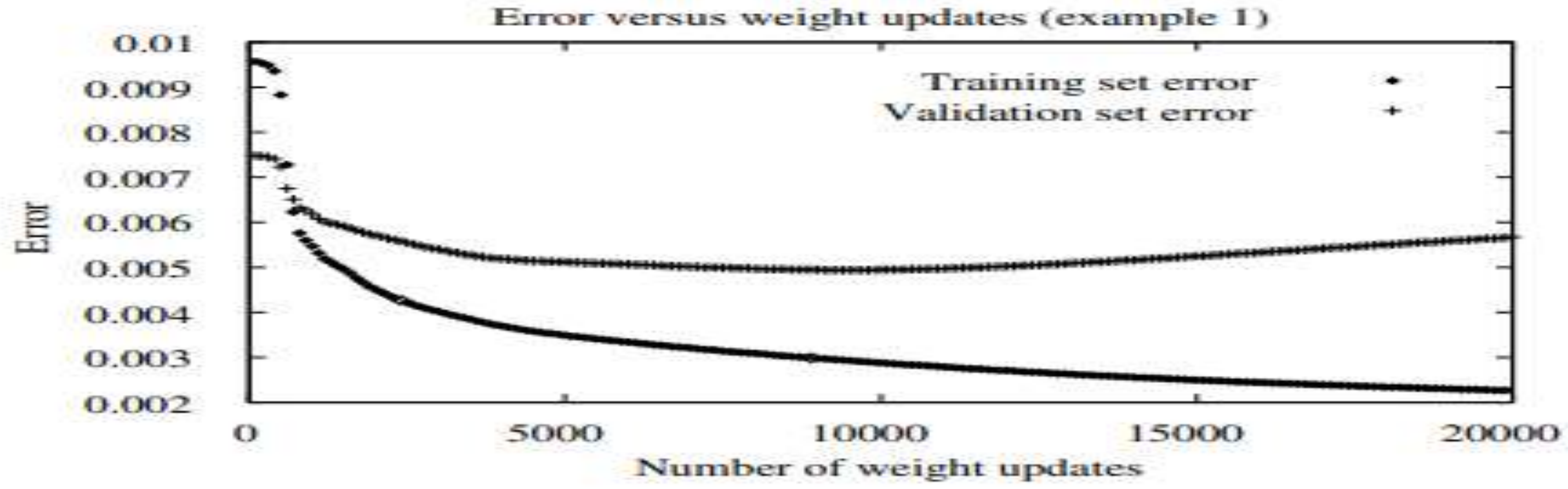
- Common heuristics to attempt to alleviate the problem of local minima include:
  - ➢ Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
  - ➢ Use stochastic gradient descent rather than true gradient descent. the stochastic approximation to gradient descent effectively descends a different error surface for each training example, relying on the average of these to approximate the gradient with respect to the full training set. These different error surfaces typically will have different local minima, making it less likely that the process will get stuck in any one of them.
  - ➢ Train multiple networks using the same data, but initializing each network with different random weights. If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected.

Generalization

- What is an appropriate condition for terminating the weight update loop?

  - One choice is to continue training until the error E on the training examples falls below some predetermined threshold.

  - this is a poor strategy because BACKPROPAGATION is susceptible to over-fitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.
- To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations. Figure 4.9 shows this variation for two fairly typical applications of BACKPROPAGATION. Consider first the top plot in this figure.
- The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples.
- This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyond the training data.



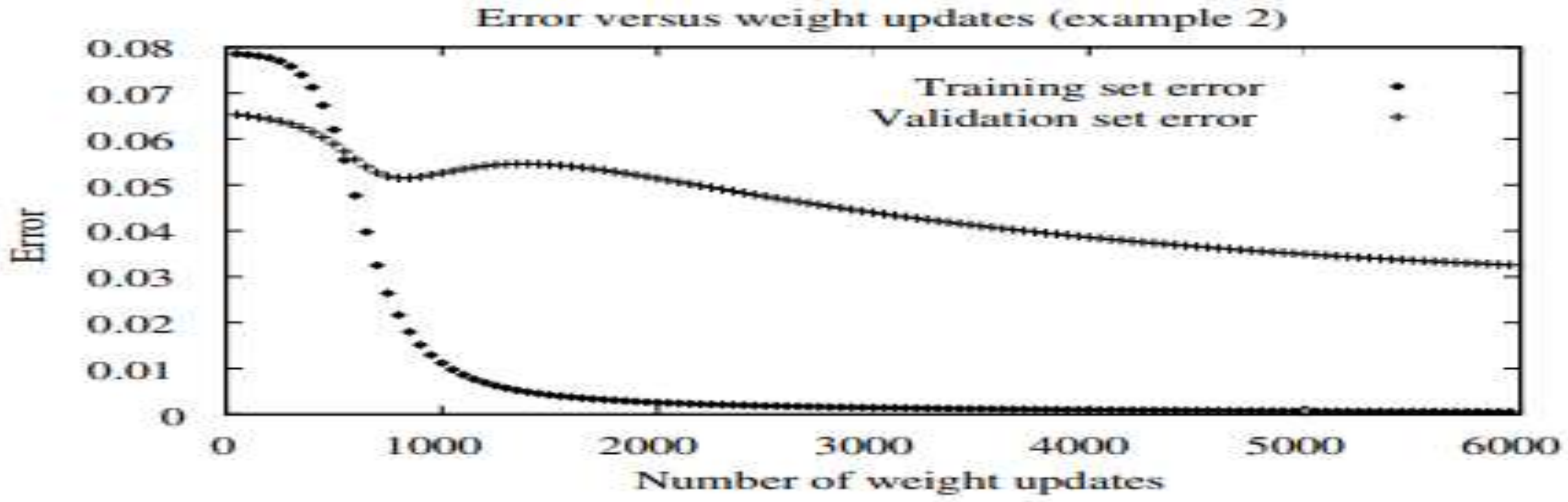


FIGURE
Plots of error E as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likely to generalize correctly to unseen data is the network with the lowest error over the validation set. Notice in the second plot, one must be careful to not stop training too soon when the validation set error begins to increase.

- generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease.
- This occurs because the weights are being tuned to fit the training examples that are not representative of the general distribution of examples.
- overfitting tend to occur during later iterations, but not during earlier iterations
- Initially network weights are initialized to small random values. With weights of nearly identical value, only very smooth decision surfaces are describable.
- As training proceeds, some weights begin to grow in order to reduce the error over the training data, and the complexity of the learned decision surface increases with the number of weight-tuning iterations.

- Several techniques are available to address the overfitting problem for **BACKPROPAGATION** learning.
- One approach, known as weight decay, is to decrease each weight by some small factor during each iteration.
- This is equivalent to modifying the definition of E to include a penalty term corresponding to the total magnitude of the network weights.
- The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.

- Unfortunately, however, the problem of overfitting is most severe for small training sets.
- In these cases, a k-fold cross-validation approach is sometimes used, in which cross validation is performed k different times,
- In one version of this approach, the m available examples are partitioned into k disjoint subsets, each of size m/k. The cross validation procedure is then run k times,
- each time using a different one of these subsets as the validation set and combining the other subsets for the training set. Thus, each example is used in the validation set for one of the experiments and in the training set for the other k - 1 experiments.
- On each experiment the cross-validation approach is used to determine the number of iterations i that yield the best performance on the validation set.
- The mean ***i*** of these estimates is then calculated, and a final run of BACKPROPAGATIO is performed ***training on*** all ***n examples*** for ***i*** iterations, with no validation set.