VIVEKANANDA
COLLEGE OF ENGINEERING & TECHNOLOGY
NEHRUNAGAR POST, PUTTUR, D.K. 574203

Lecture Notes
on

Subject Code: 15CS73
(CBCS Scheme)

Prepared by

Mr. Harivinod N
Dept. of Computer Science and Engineering,
VCET Puttur

Module-3

Artificial Neural Networks

Course website:
www.techjourney.in

Module-3: Artificial Neural Networks

1. Introduction

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the Back-propagation algorithm described in this module has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters, learning to recognize spoken words and learning to recognize faces.

Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected **neurons**. In rough analogy, 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).

To develop a feel for this analogy, let us consider a few facts from neurobiology.

The human brain, for example, 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. For example, it requires approximately 10^{-1} seconds to visually recognize your mother. Notice that the sequence of neuron firings that can take place during this 10^{-1} second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons.

This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of **highly parallel computation** based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications. While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

2. Neural Network Representations

A prototypical example of ANN learning is provided by Pomerleau's (1993) 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 30x32 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. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. 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 (driving in the left lane of a divided public highway, with other vehicles present).

Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems.

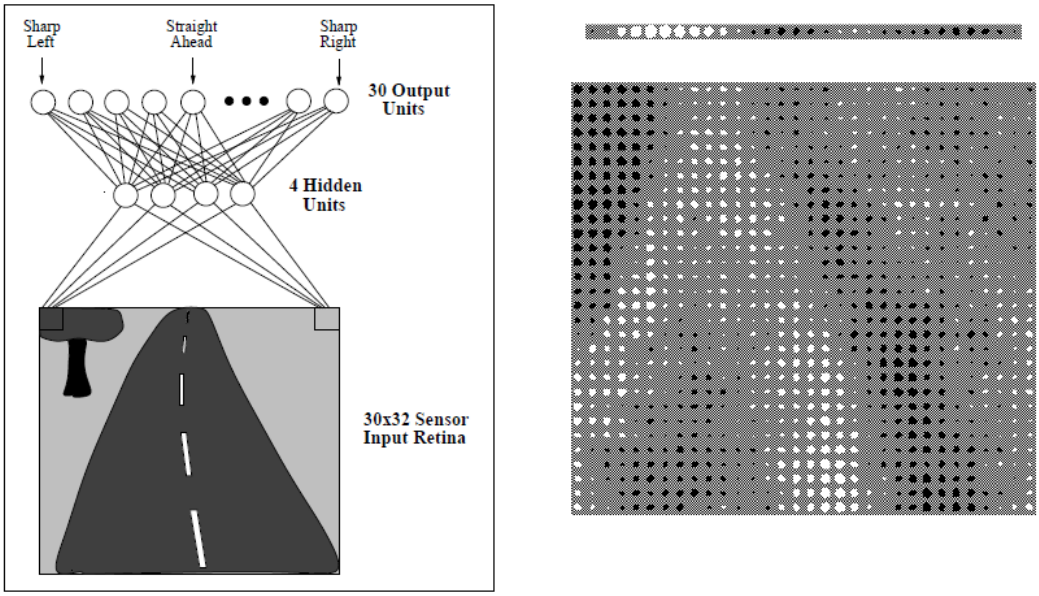


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network 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.

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. As can be seen, 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 30x32 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.

The network structure of ALYINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures-acyclic or cyclic, directed or undirected. This module will focus on the most common and practical ANN approaches, which are based on the back-propagation algorithm. The backpropagation algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

3. 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 also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. In these cases, ANN and decision tree learning often produce results of comparable accuracy. The back-propagation algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

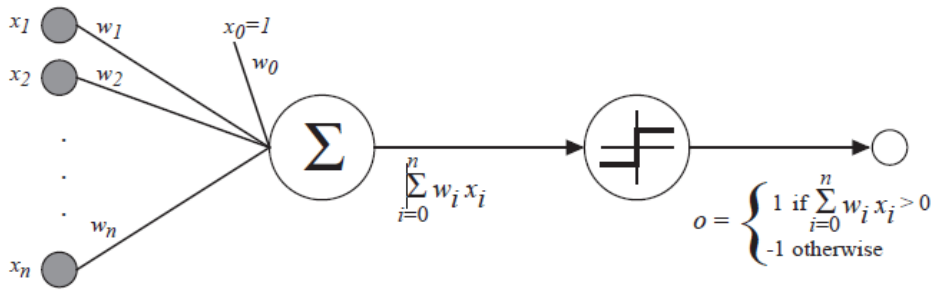
- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one 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, say, 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, and the settings of various learning algorithm parameters.
- ***Fast evaluation of the learned target function may be required.*** Although 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.

4. Perceptrons

One type of ANN system is based on a unit called a **perceptron**, illustrated in Figure given below.



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. More precisely, given inputs xl through x,, the output o(x1, . . . , x,) computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

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 ($-w_0$) is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ 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.

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

4.1 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 $\vec{w} \cdot \vec{x} = 0$. Of course, 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.

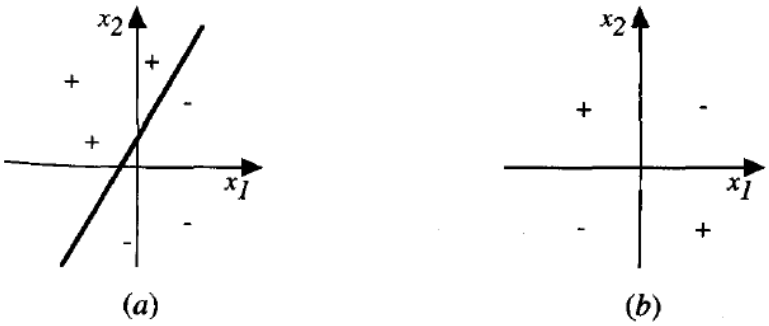


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line).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

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 wo = -3, and wl = w2 = .5. This perceptron can be made to represent the OR function instead by altering the threshold to wo = -.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 $x_1 \neq x_2$. Note the set of linearly nonseparable training examples shown in Figure 4.3(*b*) corresponds to this XOR function.

4.2. The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ±1 output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two:

1. The perceptron rule and
2. The delta rule

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.

Let us understand **perceptron rule.**

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron 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 w_i associated with input xi according to the rule

$$w_i \leftarrow w_i + \Delta w \text{ where } \Delta w_i = \eta(t - o)x_i$$

Here t is the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the 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.

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 η is used. If the data are not linearly separable, convergence is not assured.

4.3 Illustration of Perceptron training rule

Consider following training set

i/p : $x^1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}$ $x^2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$ $x^3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

Desired o/p : $t_1 = -1$ $t_2 = -1$ $t_3 = 1$

Assume learning constant $\eta = 0.1$

Weights are initialized to $w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$

Solution

Algorithm

Step 1: Compute net $= \sum w_i x_i$ [Activation fn]

Step 2: $o_i = \text{sign}(net)$ [Activation fn]

ie $o_i = \begin{cases} +1 & \text{if } x >= 0 \\ -1 & \text{if } x < 0 \end{cases}$

Step 3: Compute a) $\Delta w_i = \eta (t - o) x_i$

b) $w_i \leftarrow w_i + \Delta w_i$

Consider 1st training example

1. $net = w^T x$

$= [1, -1, 0, 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = [1 + 2 + 0 - 0.5]$

$= 2.5$

2. $o_1 = \text{sign}(2.5) = +1$

3.a) $\Delta w = \eta (t_1 - o_1) x_1$

$\Delta w = 0.1(-1 - 1) \cdot \begin{bmatrix} +1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = -0.2 \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}$

$\Delta w = \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix}$

b) $w \leftarrow w + \Delta w$

$= \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$

Consider 2nd training example

We have $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(2)} = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$ $t_2 = -1$

1. $net = w^T x$

$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$

$= [0 + (-0.9) + 0 - 0.7] = -1.6$

2. $o_2 = \text{sign}(-1.6) = -1$

3. $\Delta w = \eta (t_2 - o_2) x^{(2)} = 0$

$\therefore$ No change in w.

$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$.

Now consider 3rd example

$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(3)} = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$ $t_3 = 1$

Step 1: $net = w^T x$

$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

$= [-0.8 - 0.6 + 0 - 0.7]$

$= -2.1$

Step 2: $o_3 = \text{sign}(net)$

$= \text{sign}(-2.1) = -1$

Step 3: a) $\Delta w = \eta (t_3 - o_3) x^{(3)}$

$= 0.1(1 - (-1)) \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

$= 0.2 \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix}$

b) $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$ Final updated weights after one epoch

4.4 Gradient Descent and the Delta Rule

Although 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. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

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

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

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

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 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() is simply half the squared difference between the target output t_d and the hear unit output o_d, summed over all training examples.

Visualizing 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 w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 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.

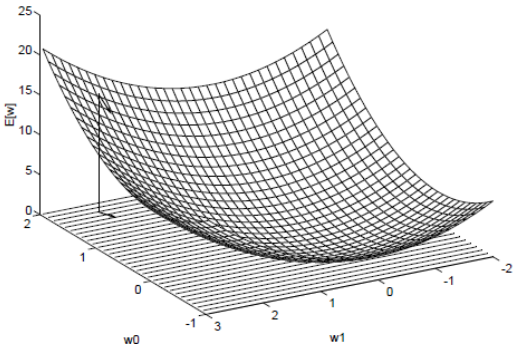


FIGURE 4.4 Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w0, w1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w0, w1 plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4. This process continues until the global minimum error is reached.

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

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \quad \text{where} \quad \Delta\vec{w} = -\eta\nabla E(\vec{w})$$

Here η 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

$$w_i \leftarrow w_i + \Delta w_i \quad \text{where} \quad \Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

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. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from Equation (4.2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned} \tag{4.6}$$

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

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

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 w_i 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 η is used. If η 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 η as the number of gradient descent steps grows.

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $< \vec{x}, t >$, where $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η is the learning rate.

- Initialize each w_i to some small random value
- Until the **termination condition** is met, Do
  - Initialize each Δw_i to zero
  - For each $< \vec{x}, t >$ 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 $\Delta w_i = \Delta w_i + \eta(t - o)x_i^*$
  - For each linear unit weight w_i, Do $w_i \leftarrow w_i + \Delta w_i^{**}$

To implement incremental approximation, equation ** is deleted and equation * is replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Table 4.1: Gradient Descent Algorithm for training a linear unit.

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

$$\Delta w_i = \eta(t - o)\, x_i \qquad (4.10)$$ (Delta Rule, also called as LMS least mean square)

where t, o, and x_i are the target value, unit output, and i^{th} input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation marked with ** is simply deleted and Equation marked with * 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

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \qquad (4.11)$$

where t_d, 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 $E(\vec{w})$. 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 key differences are listed below.

Standard gradient descent	***Stochastic gradient descent***
1. Error is summed over all examples before updating weights	1. Weights are updated upon examining each training example
2. Requires more computation per weight update step	2. Require less computation
3. Converges to local minima	3. Sometimes avoid falling into these local minima

4.5 Remarks

We have considered two similar algorithms for iteratively learning perceptron weights. The key difference between these algorithms are listed below

Perceptron training rule	***Delta rule***

1.Updates weights based on the error in the thresholded perceptron output	1. Updates weights based on the error in the un-thresholded linear combination of inputs
2.converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.	2. converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.

5. Multilayer Networks and The Backpropagation 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.

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.

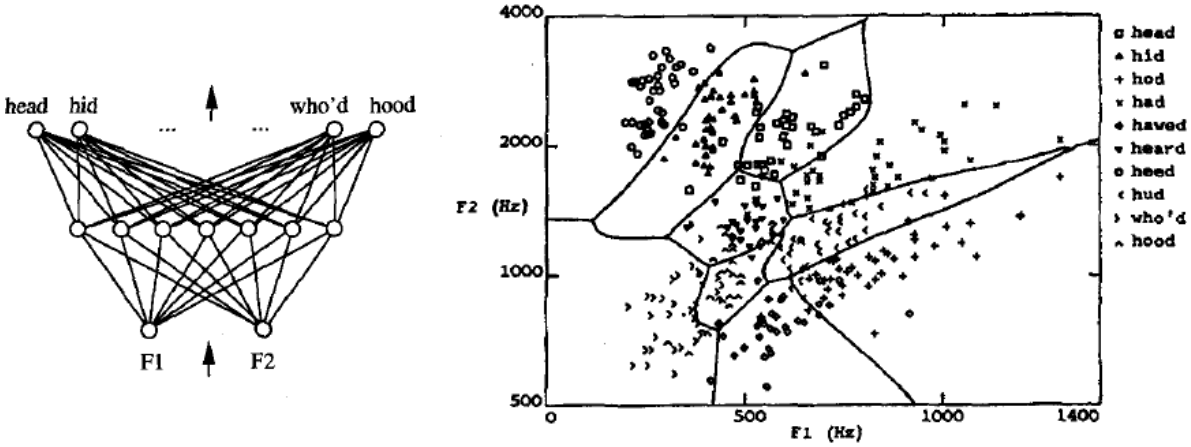


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network 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).)

This section discusses how to learn such multilayer networks using a gradient descent algorithm

5.1 A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable 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.

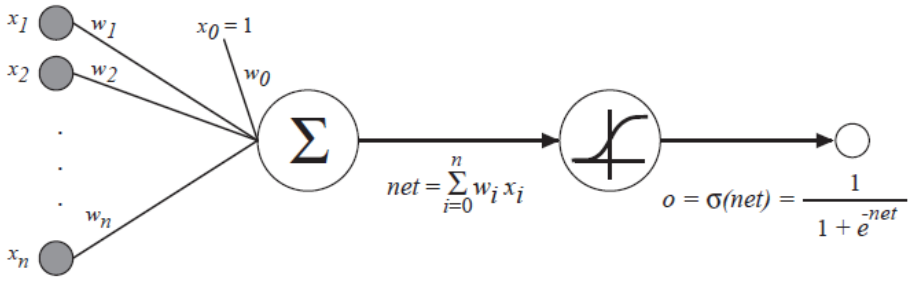


Figure 4.6: A sigmoid threshold unit

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. More precisely, the sigmoid unit computes its output o as

$$o = \sigma(\vec{w} \cdot \vec{x}) \text{ where } \quad \sigma(y) = \frac{1}{1 + e^{-y}}$$

σ is often called the sigmoid function or, alternatively, the logistic function. Note 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 squashing function of the unit.

The sigmoid function has the useful property that its derivative is easily expressed in terms of its output. σ'(y) = σ(y) (1- σ(y))

5.2 The Backpropagation Algorithm

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 are 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. The learning problem faced by Backpropagation search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar

to that shown for linear units in Figure 4.4. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

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.

η 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 δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

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. 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 Algorithm been found to produce excellent results in many real-world applications.

The Backpropagation Algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward 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.
- x_{ji} denotes the input from node i to unit j, and w_{ji} 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, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 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 (Equation [T4.5]) is similar to the delta training rule. Like the delta rule, it updates each weight in proportion to the learning rate η, the input value x_{ji} 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_k is computed for each network output unit k (Equation [T4.3]). δ_k is simply the familiar $(t_k - o_k)$ 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 has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values t_k 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 terms Jk for each output unit influenced by h, weighting each of the δ_k's by w_{kh}, 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.

The algorithm updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the $\delta_j x_{ji}$ values over all training examples before altering weight values.

The weight-update loop in Backpropagation may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data.

5.3 Learning in Arbitrary Acyclic Networks

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the 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

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr}\, \delta_s \tag{4.19}$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

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

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\, \delta_s \tag{4.20}$$

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. It is this general form of the weight-update rule that we derive in next Section.

5.4 Derivation of the Backpropagation rule

This section presents the derivation of the BACKPROPAGATION weight-tuning rule. It may be skipped on a first reading, without loss of continuity.

The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm in Table 4.2. Recall from Equation (4.11) that stochastic gradient descent involves iterating through the training examples one at a time, 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

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

Here *outputs* is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation shown in Figure 4.6, adding a subscript j to denote to the jth unit of the network as follows:

- 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 seen in Equation (4.21). 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

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \tag{4.22}$$

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 $k = j$.

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

$$= o_j(1 - o_j) \tag{4.25}$$

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

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

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

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji} \tag{4.27}$$

Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm of Table 4.2. Furthermore, we can see now that δ_k in Equation (T4.3) is equal to the quantity $-\frac{\partial E_d}{\partial net_k}$. In the remainder of this section we will use δ_i to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for an arbitrary unit i.

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 w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by $Downstream(j)$. Notice that net_j can influence the network outputs (and therefore E_d) 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

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

which is precisely the general rule from Equation (4.20) for updating internal unit weights in arbitrary acyclic directed graphs. Notice Equation (T4.4) from Table 4.2 is just a special case of this rule, in which $Downstream(j) = outputs$.

Illustration

Refer Additional notes provided in Techjoourney.in

6. Remarks on The Backpropagation algorithm

6.1 Convergence and Local Minima

As shown above, 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, it is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Despite the lack of assured convergence to the global minimum error, backpropagation is a highly effective function approximation method in practice. In many practical applications the problem of local minima has not been found to be as severe as one might fear. 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. Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions. One might expect more local minima to exist in the region of the weight space that represents these more complex functions. One hopes that by the time the weights reach this point they have already moved close enough to the global minimum that even local minima in this region are acceptable.

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

- Add a momentum term to the weight-update rule
- Use stochastic gradient descent rather than true gradient descent.
- Train multiple networks using the same data, but initializing each network with different random weights.

6.2 Representational Power of Feedforward Networks

What set of functions can be represented by feedforward networks? Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known:

- Boolean functions. Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.
- Continuous functions. Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units. The networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer

will achieve this. The number of hidden units required depends on the function to be approximated.

- Arbitrary functions. Any function can be approximated to arbitrary accuracy by a network with three layers of units. Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general.

6.3 Hypothesis Space Search and Inductive Bias

The hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis. This structure is quite different from the general-to-specific ordering algorithms, or the simple-to-complex ordering over decision trees algorithms.

What is the inductive bias by which backpropagation generalizes beyond the observed data? It is difficult to characterize precisely the inductive bias of backpropagation, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points. Given two positive training examples with no negative examples between them, backpropagation t end to label points in between as positive examples as well.

6.4 Hidden Layer Representations

One intriguing property of Backpropagation is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead Backpropagation to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

The ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning. In contrast to learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer.

6.4 Generalization, Overfitting, and Stopping Criterion

In the Backpropagation algorithm, the termination condition for the algorithm has been left unspecified. What is an appropriate condition for termination the weight update loop? One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold. In fact, this is a poor strategy because Backpropagation is

susceptible to overfitting 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 Backpropogation.

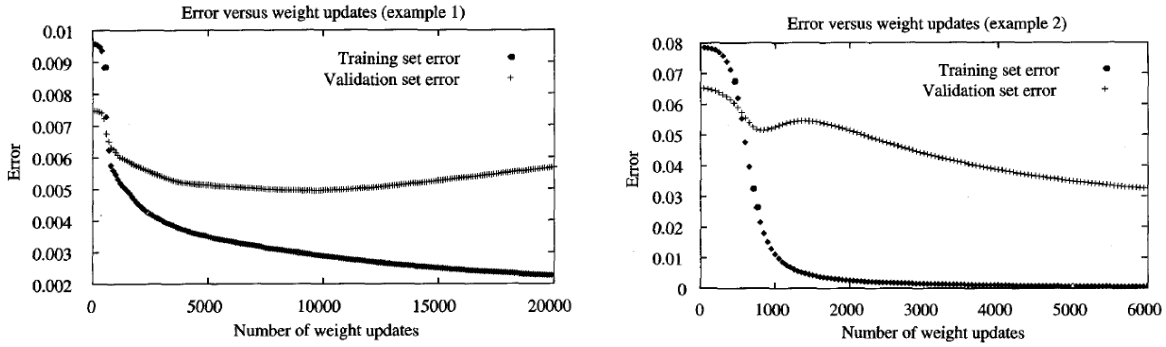


FIGURE 4.9 Plots of error E as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likeIy 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.

Consider first 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

Why does overfitting tend to occur during later iterations, but not during earlier iterations? Consider that 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. Thus, the effective complexity of the hypotheses that can be reached by Backpropagation increases with the number of weight-tuning iterations. This overfitting problem is analogous to the overfitting problem in decision tree learning.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search.

How many weight-tuning iterations should the algorithm perform? Clearly, it should use the number of iterations that produces the lowest error over the validation set, since this is the best indicator of network performance over unseen examples. In typical implementations of this approach, two copies of the network weights are kept: one copy for training and a separate copy of the best-performing weights thus far, measured by their error over the validation set. Once the trained weights reach a significantly higher error over the validation set than the stored weights, training is terminated and the stored weights are returned as the final hypothesis.

.

7. Summary

Main points of this chapter include:

- Artificial neural network learning provides a practical method for learning real-valued and vector-valued functions over continuous and discrete-valued attributes, in a way that is robust to noise in the training data. The Backpropagation algorithm is the most common network learning method and has been successfully applied to a variety of learning tasks, such as handwriting recognition and robot control.
- The hypothesis space considered by the Backpropagation algorithm is the space of all functions that can be represented by assigning weights to the given, fixed network of interconnected units. Feedforward networks containing three layers of units are able to approximate any function to arbitrary accuracy, given a sufficient (potentially very large) number of units in each layer. Even networks of practical size are capable of representing a rich space of highly nonlinear functions, making feedforward networks a good choice for learning discrete and continuous functions whose general form is unknown in advance.
- Backpropagation searches the space of possible hypotheses using gradient descent to iteratively reduce the error in the network fit to the training examples. Gradient descent converges to a local minimum in the training error with respect to the network weights. More generally, gradient descent is a potentially useful method for searching many continuously parameterized hypothesis spaces where the training error is a differentiable function of hypothesis parameters.
- One of the most intriguing properties of Backpropagation is its ability to invent new features that are not explicit in the input to the network. In particular, the internal (hidden) layers of multilayer networks learn to represent intermediate features that are useful for learning the target function and that are only implicit in the network inputs.
- Overfitting the training data is an important issue in ANN learning. Overfitting results in networks that generalize poorly to new data despite excellent performance over the training data. Cross-validation methods can be used to estimate an appropriate stopping point for gradient descent search and thus to minimize the risk of overfitting.

*****