# MODULE 3: ARTIFICIAL NEURAL NETWORKS



| References |
|---|
| 1. Tom M. Mitchell, Machine Learning, India Edition 2013, McGraw Hill Education. |

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPOGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN 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.

## 3.1 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 BACKPROPAGATION algorithm has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989).

### 3.1.1 Biological Motivation

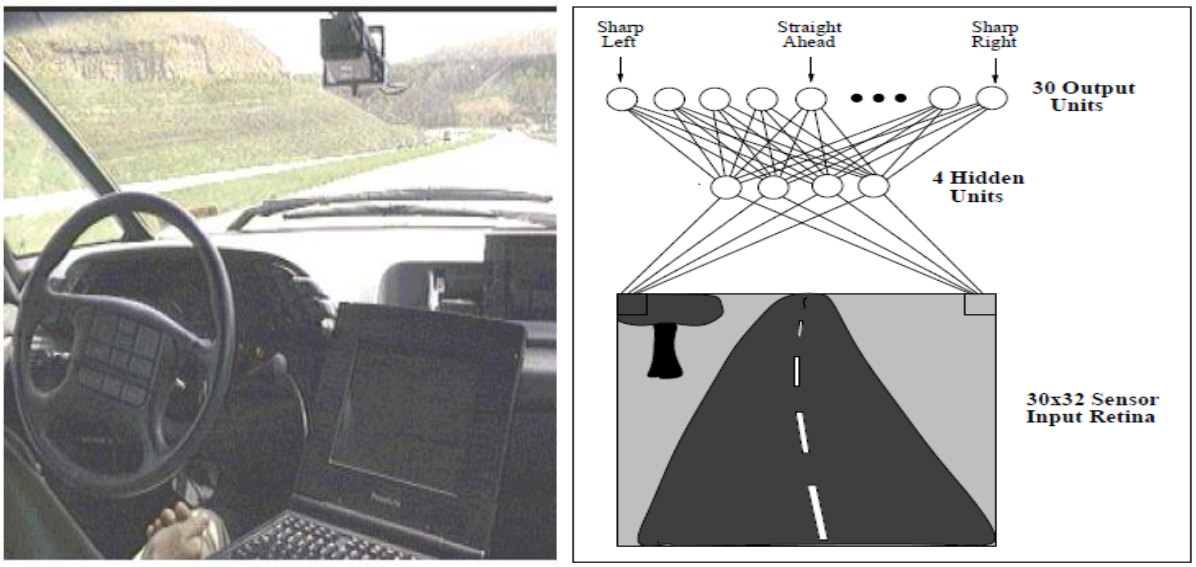
The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each 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.

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

## 3.2 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 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. 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).



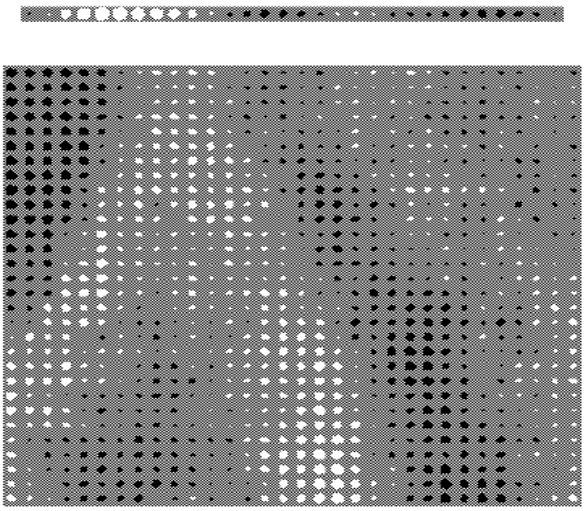



**FIGURE 3.1:** Neural network learning to steer an autonomous vehicle. The ALVINN system uses **BACKPROPAGATION** to learn to steer an autonomous vehicle (photo at top left) driving at speeds up to **70** miles per hour. The diagram on the top right shows how the image of a forward-mounted camera is 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 bottom left shows weight values for one of the hidden units in this network. The **30** x 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.

## 3.3 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. The BACKPROPAGATION 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.
- ***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.

## 3.4 PERCEPTRON

One type of ANN system is based on a unit called a perceptron, illustrated in Figure 3.2. 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.

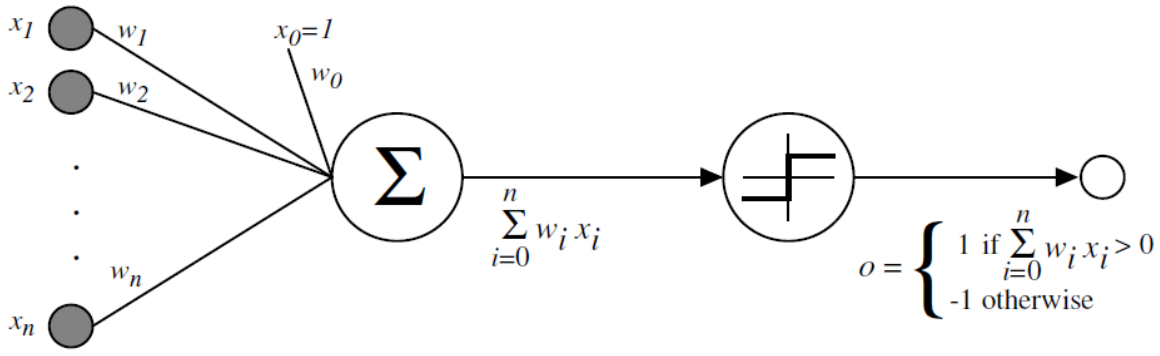



**Figure 3.2: A perceptron**

More precisely, given inputs ***x1*** through ***xn*** the output ***o(x1,*** . . . , ***xn)*** 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 ***xi*** 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 ***xo*** = 1, allowing us to write the above inequality a $\sum_{i=0}^{n} w_i x_i > 0$ or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

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

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

### 3.4.1 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 3.3.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.

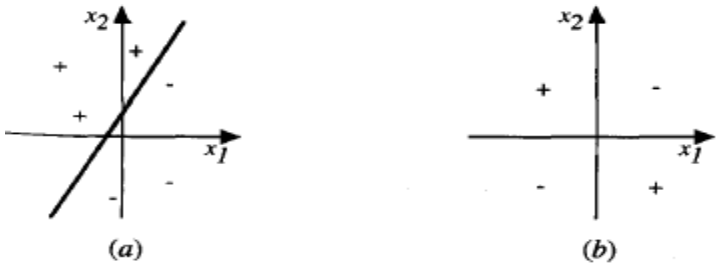



**Figure 3.3: The decision surface represented by a two-input perceptron. *(a)* A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). *X1* and *X2* 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* = *-.8,* and *w1* = *w2* = *.5*. This perceptron can be made to represent the OR function instead by altering the threshold to *wo* = *-.3*.

### 3.4.2 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 ± output for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta 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 ***wi*** associated with input ***xi*** according to the rule:

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

where

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

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and *q* 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.

### 3.4.3 Gradient Descent and the Delta Rule

The perceptron rule will fail to converge if the examples are not linearly separable. The second rule called 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. The delta training rule is best understood by considering the task of training an unthreshold perceptron; that is a linear unit for which the output O is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \quad \text{------- (1)}$$

The measure for the ***training error*** of a hypothesis (weight vector)

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

### 3.4.4 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 3.4

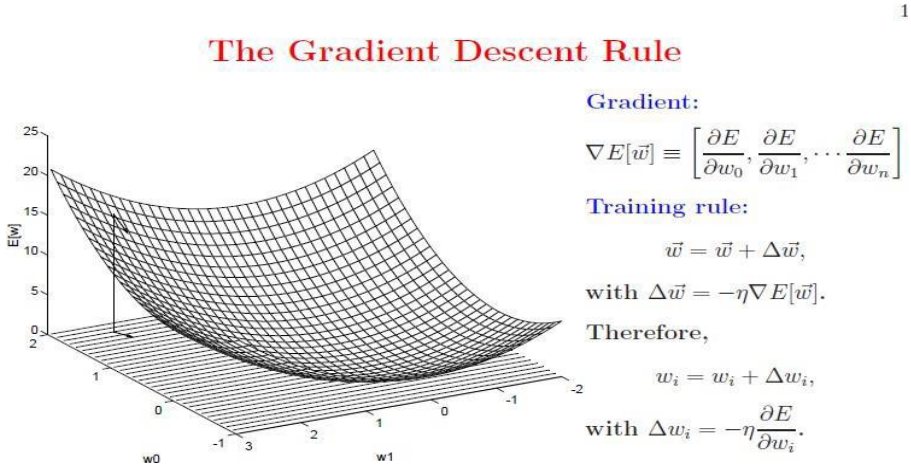



**Figure 3.4: Hypothesis Space**

Here the axes $w_o$ and $w_1$ represent possible values for the two weights of a simple linear unit. The $w_o$, $w_1$ 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.

### 3.4.5 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 $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \quad \text{----- (3)}$$

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 3.5 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the $w_o$, $w_1$ 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

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \quad \text{---------- (4)}$$

Here $\eta$ 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

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad \text{---------- (5)}$$

To construct a practical algorithm for iteratively updating weights according to Equation 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 2, as

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

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

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

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

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \qquad \text{------- (6)}$$

where $x_{id}$ denotes the single input component $x_i$ for training example d. We now have an equation that gives in terms of the linear unit inputs $x_{id}$, outputs $O_d$, and target values $t_d$ associated with the training examples. Substituting Equation 6 into Equation 5 yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \qquad \text{-------- (7)}$$

### 3.4.6 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.

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,* $t$ *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

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

  - For each linear unit weight $w_i$, Do

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

**Table 1: Gradient Descent algorithm for training a linear unit**

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 **(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 **(7)** except that ***as*** we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\ x_i \quad \text{---------- (8)}$$

where **t, o,** and **xi** are the target value, unit output, and $i^{th}$ input for the training example in question. To modify the gradient descent algorithm of Table **1** to implement this stochastic approximation, Equation **1** is simply deleted and is 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

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \quad \text{---------- (9)}$$

where $\mathbf{t_d}$, 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 $E_d(\vec{w})$.

## 3.5 MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Single perceptron's 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 3.5.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.).

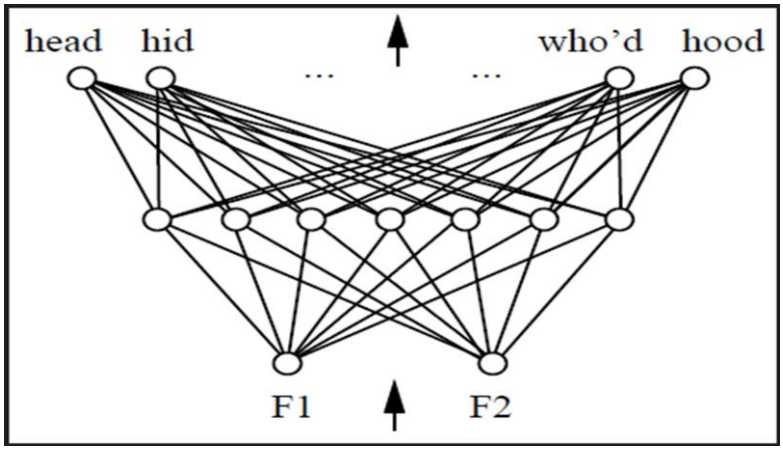



**Figure 3.5: A typical multilayer network**

### 3.5.1 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.

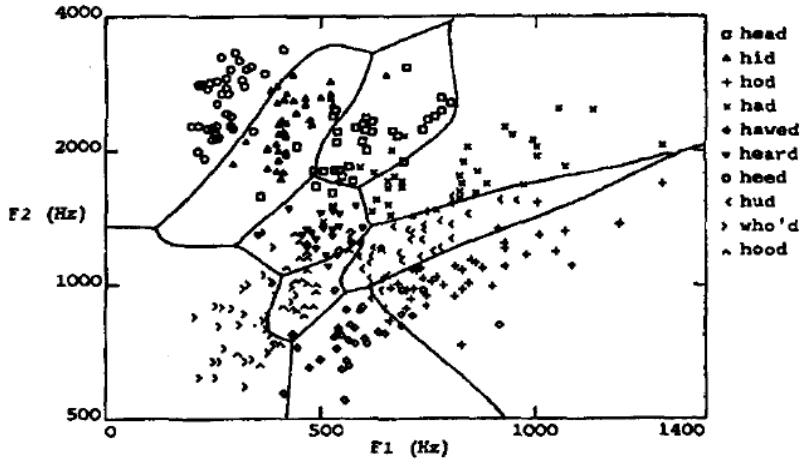



**Figure 3.6: Decision regions of a multilayer feed-forward network**

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. The sigmoid unit is illustrated in Figure 3.7.

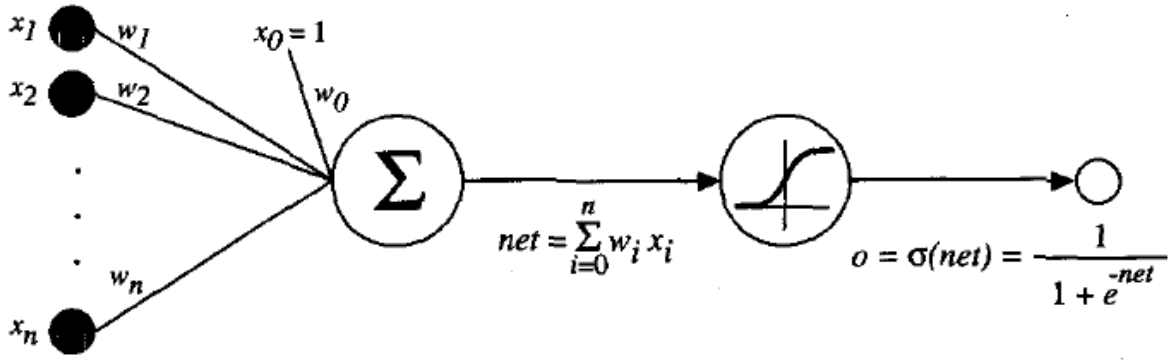



**Figure 3.7: The sigmoid threshold unit**

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

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

Where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \quad \text{---------- (10)}$$

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.

### 3.5.2 The 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. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION. 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

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{---------- (11)}$$

where ***outputs*** is the set of output units in the network, and $\mathbf{t_{kd}}$ and $\mathbf{o_{kd}}$ are the target and output values associated with the k[th] 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* $(\vec{x}, \vec{t})$, *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 $(\vec{x}, \vec{t})$ 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 $\delta_k$

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

3. For each hidden unit $h$, calculate its error term $\delta_h$

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

4. Update each network weight $w_{ji}$

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

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \quad \text{(T4.5)}$$

**Table 2: The stochastic gradient descent version of the Backpropagation algorithm for feed-forward networks containing two layers of sigmoid units.**

The **Backpropagation algorithm** is presented in Table 2. 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 algorithm. 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.
- $\mathbf{x_{ji}}$ denotes the input from node i to unit ***j*** **,** and $\boldsymbol{w_{ji}}$ denotes the corresponding weight.

  $\delta_n$ 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,

$$\delta_n = -\frac{\partial E}{\partial net_n}$$

### 3.5.3 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 2. Recall from Equation **(10)** 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 ***Ed*** with respect to this single example. In other words, for each training example d every weight ***wji*** is updated by adding to it $\Delta w_{ji}$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{---------- (12)}$$

where ***Ed*** is the error on training example d, summed overall 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, ***tk*** is the target value of unit k for training example d, and ***ok*** 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 3.7, adding a subscript ***j*** to denote to the jth unit of the network as follows:

- $x_{ji}$ = the $i$th input to unit $j$
- $w_{ji}$ = the weight associated with the $i$th 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 $j$
- $\sigma$ = 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 ***12.***

### Stage/Case 1: Computing the increments ($\Delta$) for output unit weights

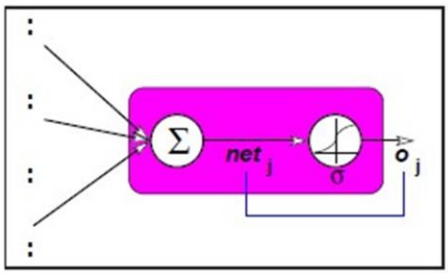
$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1-o_j)$$

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

$$= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$

$$= -(t_j - o_j)$$




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

$$\Rightarrow \delta_j \overset{not.}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)(t_j - o_j)$$

$$\Rightarrow \Delta w_{ji} = \eta\delta_j x_{ji} = \eta o_j(1-o_j)(t_j - o_j)x_{ji}$$

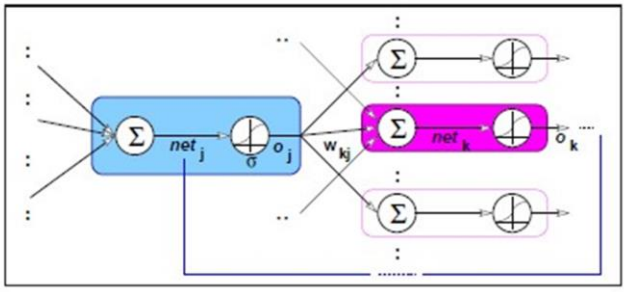
### Stage/Case 2: Computing the increments ($\Delta$) for hidden unit weights

$$\frac{\partial E_d}{\partial net_j} = \sum_{k\in Downstream(j)}\frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k\frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k\frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k w_{kj}\frac{\partial o_j}{\partial net_j} = \sum_{k\in Downstream(j)} -\delta_k w_{kj}o_j(1-o_j)$$




**Therefore:**

$$\delta_j \overset{not}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}$$

$$\Delta w_{ji} \overset{def}{=} -\eta\frac{\partial E_d}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}x_{ji} = \eta\delta_j x_{ji} = \eta\left[\, o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}\,\right]x_{ji}$$

**Acknowledgement**

The diagrams and tables are taken from the textbooks specified in the references section.

**Prepared by:**

Rakshith M D

Department of CS&E

SDMIT, Ujire