R.J. Marks II, "Artificial Neural Networks: Supervised Learning", in Artificial Neural Networks with Applications to Power Systems, M.A. El-Sharkawi & Dagmar Niebur, Editors, IEEE, PES Tutorial, 1996 (IEEE Press)

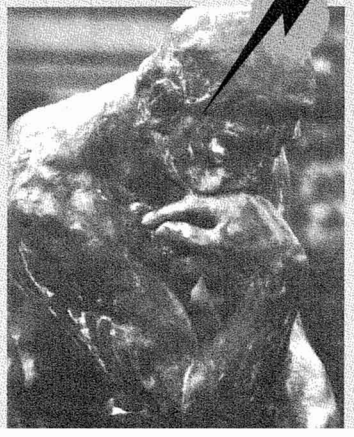
IEEE

Power Engineering Society

A Tutorial Course on

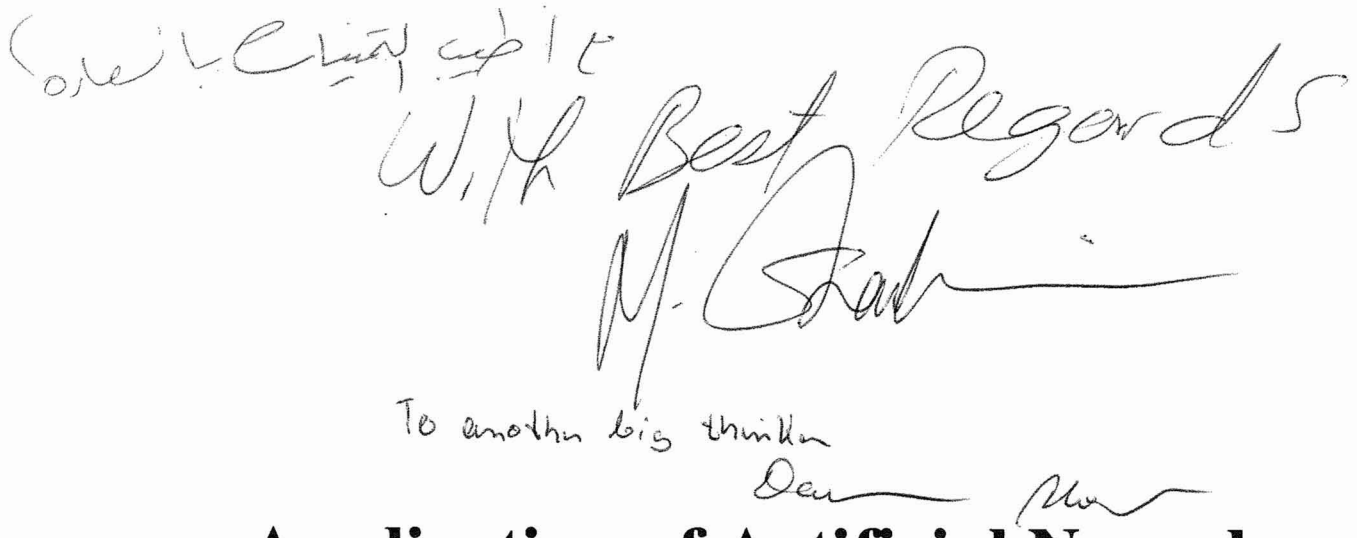
Artificial Neural Networks with Applications to Power Systems

Edited by
Mohamed El-Sharkawi
and
Dagmar Niebur



96 TP 112-0

POWER ENGINEERING SOCIETY
PES
IEEE



Application of Artificial Neural Networks to Power Systems

Sponsored by:

Intelligent Systems Applications Working Group

Computer and Analytical Methods Subcommittee

Power System Engineering Committee

Edited by

M. A. El-Sharkawi

and

Dagmar Niebur

Chapter 3
Artificial Neural Networks: Supervised Models

3.0. INTRODUCTION

Supervised learning attempts to determine the relationship between a set of input or stimulus data and corresponding output data. The learning is *supervised* because when the student system offers a response to a given input, a teacher, or *supervisor*, is able to present the true output. By comparing the system response to the true output, the student is better able to properly respond to the input when subsequently presented.[1]

Although there exists a number of paradigms for supervised learning, neural networks have been shown to be quite adept at learning the relationship among input-output data.

3.1. THE LAYERED PERCEPTRON

The most commonly used artificial neural network is the *layered perceptron* shown in Figure 3.1. The solid dots at the middle (hidden) and top layers are *neurons*. The linkage between neurons are *interconnects*. The biological counterpart of the interconnect is the *synapse*. Artificial neural networks have been claimed to emulate their biological counterpart. Any resemblance is, at best, crude.

Each neuron has a *state* which is a scalar number. The state of a neuron is determined by the interconnects to and states of the neurons that feed it. This is illustrated in Figure 3.2. Here, the state, s_j, of a neuron is determined by the states, $\{s_k | 1 \leq k \leq K_{\ell-1}\}$, of all of the neurons below it where K_ℓ is the number of neurons in layer ℓ. (In Figure 3.2, $K_{\ell-1} = 5$). The weight of interconnect between neurons s_k and neuron s_j is w_{jk}. The sum of the weighted input neurons is

$$\text{sum}_j = \sum_{k=1}^{K} w_{jk} s_k \tag{3.1}$$

Note this is simply a matrix-vector multiplication. This sum is made the argument of a function $\sigma(\cdot)$. A commonly used nonlinearity is the *sigmoid*

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

[1] For *unsupervised* learning, there is no teacher. The learning system in this case clusters the input data into various categories.

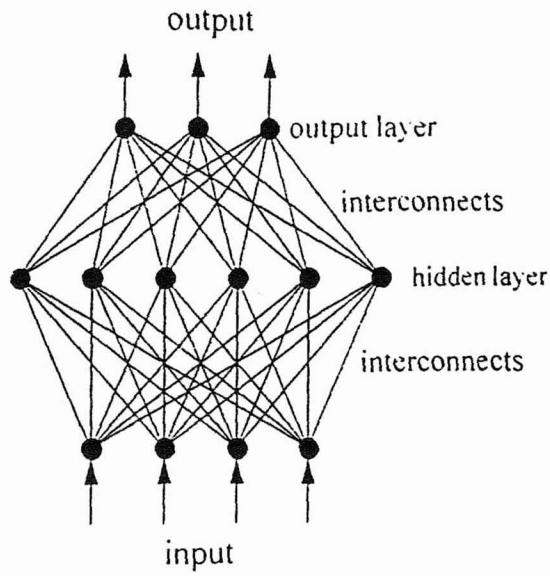


Figure 3.1 A layered perceptron neural network with one hidden layer.

As is illustrated in Figure 3.3, the sigmoid monotonically maps the interval $-\infty < x < \infty$ into the interval $0 < \sigma(x) < 1$. Other *squashing functions*, such as a hyperbolic tangent, may be used. As will be shown, one advantage of the sigmoid in Equation 3.2 is

$$\frac{d\sigma(x)}{dx} = \sigma(1 - \sigma) \tag{3.3}$$

3.2. LAYERED PERCEPTRON NEURAL NETWORK TRAINING

The layered perceptron's job, using training data, is to learn the relationship between a stimulus and a response. The training data is in two pieces - the input and output. Let there be N components in the training set. The input vectors are

$$\{\vec{i}_n | 1 \leq n \leq N\}$$

and the output, or *target* vectors are

$$\{\vec{t}_n | 1 \leq n \leq N\}.$$

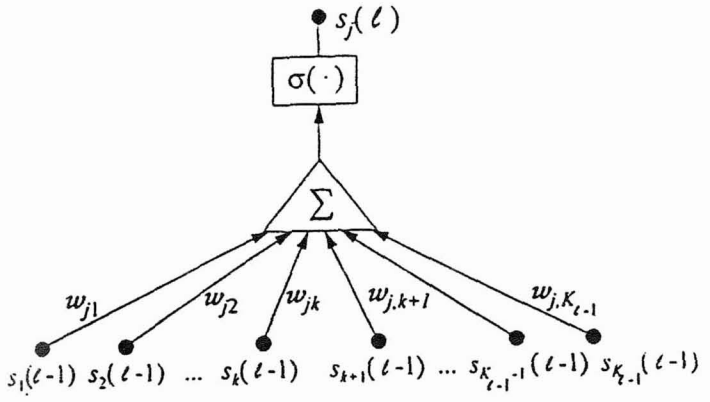


Figure3.2 The state of the jth neuron in layer ℓ is denoted by $s_j(\ell)$. It is determined by the states of the neurons in the layer below, $\{s_k(\ell-1)|1 \leq k \leq K_{\ell-1}\}$, where $K_{\ell-1}$, the number of neurons in layer $(\ell-1)$, is five in this figure. The states of the neurons in layer $(\ell-1)$ are weighted by the interconnect weights, w_{jk}, and the results added to give the number sum_j. This number is passed through a nonlinearity, $\sigma(\cdot)$, to give the desired neural state.

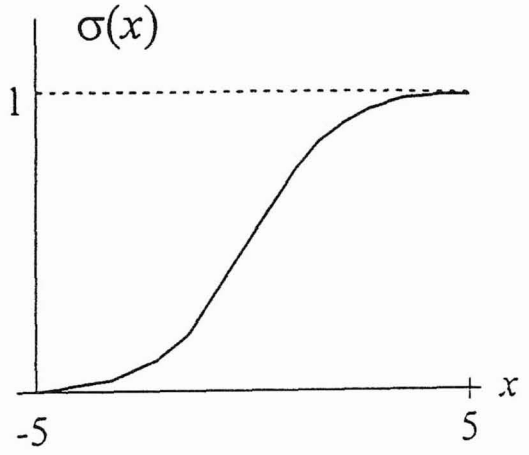


Figure3.3 The sigmoid, $\sigma(x) = [1+\exp(-x)]^{-1}$, 'squashes the range over all of x into the interval (0,1).

The input $\vec{i}_1$ is associated with a response of $\vec{t}_1$, $\vec{i}_2$ with $\vec{t}_2$, etc. It is convenient to interpret the layered perceptron to be successfully trained when an input of $\vec{i}_n$ produces states equal to $\vec{t}_n$ at the output layer[2].

To pedagogically illustrate, suppose we wish to train the neural network to distinguish between books and magazines. The $\vec{i}_n$ vectors would contain digital information on features that distinguish magazines from books. The first element of $\vec{i}_n$ could, for example, be a one if the cover of the book/magazine was hard cardboard and zero if was made of flexible paper. This *feature* is, of course, insufficient to classify books from magazines, since many books (*eg.* paperback books) do not have hard covers. From the observation that books usually have more pages, a second feature might be the number of pages in the book/magazine. The third feature might indicate whether staples or glue is used to bind the magazine/book, etc. We indicate books by logic one and magazines by a zero. Thus, if $\vec{i}_1$ were the features of a book, then $\vec{t}_1 = t_1 = 1$, (*i.e* the output in this example is scalar). If $\vec{i}_2$ were also the features of a book, then $t_2 = 1$. An output of $t_3 = 0$ is appropriate if the input vector $\vec{i}_3$ has features corresponding to a magazine, etc.

The neural network can be *trained* by repeatedly exposing the network to the training data. In one popularly used method, the input $\vec{i}_1$, corresponding to a book, is imposed on the input. A value of one is desired at the output. The output, however, will be other than one. The weights of the neural network are adjusted so that the next time the neural network sees $\vec{i}_1$, it will have an output which is a bit closed to one. The input $\vec{i}_2$ is placed on the input and an output of one is again imposed. The weights are then adjusted slightly to appropriately respond to $\vec{i}_2$. The process is repeated using the magazine data vector $\vec{i}_3$ with an output of zero. By the time we reach the last vector, $\vec{i}_N$, the first adjustment corresponding to $\vec{i}_1$ has been largely 'forgotten' by the neural network. Therefore, the process is repeated until the weights of the neural network stabilize. If the neural network is trained properly, it will properly categorize a feature vector it has not seen before as a book or a magazine.

Although this description is oversimplified, it captures the essence of training a layered perceptron neural network. The training is referred to as *supervised* because, in each instance, the proper categorization of the input vector is specified. Thus, the neural network has a teacher or supervisor to tell it what each of the input training vectors is.

The example of distinguishing between books and magazines is a classification problem. Artificial neural networks can also be trained to provide continuous value regression type responses, such as forecasting. Suppose, for example, one wished to forecast 4:00 PM average speed of traffic near downtown Seattle on Interstate 5 at 3:00 PM. For feature data, we would choose data available to us at 3:00 PM, such as the day of the week, the current average speed, the number of cars per second and whether or not there was a sporting event, such as a Mariners or Seahawks game, at the Kingdome. Historical data would be used to train the net. In such cases, the average car speed at 4:00 PM is known and is the used as the output

[2]Proper training , in many cases, requires that this relationship not be learned exactly. If so, the neural network can be *memorizing* rather than *learning*. This point will be discussed later

of the neural network in training. If successfully trained, the neural network, when presented with the feature data at 3:00 PM would accurately forecast the average traffic average traffic speed.

3.2.1. ERROR BACKPROPAGATION

The most popular method of training a layered perceptron neural network is through a procedure called *error backpropagation* or BP. We will illustrate BP using the multilayer perceptron pictured in Figure 3.4. The neural net has L layers of neurons. The input is not counted as a layer since the nodes do no computation. The state of the neurons in the ℓth layer is denoted by the vector $\vec{s}(\ell)$ the jth component of which is $s_j(\ell)$. The interconnect from the kth neuron in the $(\ell-1)$st layer to the jth neuron in the ℓth layer is $w_{jk}(\ell)$. The sum of the inputs into the jth neuron in the ℓth layer is

$$\text{sum}_j(\ell) = \sum_{k \in \text{layer } (\ell-1)} w_{jki}(\ell) s_k(\ell-1). \quad (3.4)$$

The corresponding state of the jth neuron in the ℓth layer is

$$s_j(\ell) = \sigma\left(\text{sum}_j(\ell)\right)$$

where σ is a nonlinearity such as the sigmoid in Equation 3.2. The output of the neural network is equal to the states of the output neurons

$$\vec{o} = \vec{s}(L).$$

For a given set of weights, the input-output relationship of the neural network can be expressed simply by the function $\vec{o}(\vec{i})$. For a given training data set, a successfully trained neural network should roughly satisfy the relation

$$\vec{o}(\vec{i}_n) \approx \vec{t}_n.$$

In other words, the response of the neural network to $\vec{i}_n$ should be roughly equal to $\vec{t}_n$. The corresponding error is

$$E_n = \frac{1}{2}||\vec{o}(\vec{i}_n) - \vec{t}_n||^2, \quad (3.5)$$

where the (mean square) norm of a vector is defined by

$$||\vec{v}||^2 = \vec{v}^T \vec{v},$$

and the superscript T denotes transposition. Dropping the n subscript in Equation 3.5, an equivalent expression is

$$E = \frac{1}{2}\sum_{m=1}^{M}(o_m - t_m)^2 \quad (3.6)$$

where o_m and t_m are the mth components of the vectors $\vec{o}_n$ and $\vec{t}_n$ and M is the dimension of the output vector.[3]

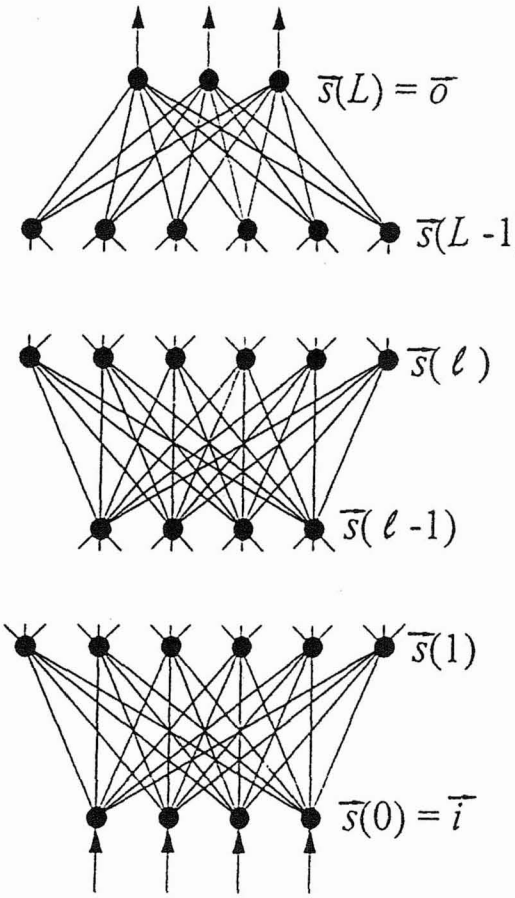


Figure 3.4 A layered perceptron with L layers. The 'zeroth' layer is the input, $\vec{i}$. The states of the neurons in the ℓth level are in the vector $\vec{s}(\ell)$. The weights of the interconnects between the kth neuron in layer $(\ell-1)$. The output corresponds to the weights in the last or Lth level. In other words, $\vec{o} = \vec{s}_L$.

Since the output of the neural network is a function of the weights of the interconnect, so is the error. Training a neural network consists of finding the weights that appropriately minimize the error. Doing so amounts to a classic search problem. There exist numerous ways to do so. The most common technique of layered perceptron training is using error backpropagation. The advantage of such an approach is that all of the training operations can be performed internal to the neural network architecture. Other search techniques are applied only to the mathematics describing the operation of the layered perceptron without regard to its architecture.

[3] In Figure 3.4. $M = 3$.

3.2.2. STEEPEST DESCENT SEARCH

Error backpropagation is a special case of steepest descent search - also referred to as the Widrow-Hoff algorithm. The error, E, is a function of the weights of the neural network. Envision this function as a two dimensional surface with hills and valleys. The desired minimum lies at the bottom of the valley. The current set of neural network weights do not put us at this desired point. We wish to take a step in the direction of minimum location. A good guess is to step in the direction where the slope is downhill. In the new location, the direction of downhill is again chosen and another step taken. Eventually, the hope is that this repeated procedure will result at being at the minimum. If the choice is made in each step to proceed in the direction where this slope is maximum, the resulting search technique is referred to as *steepest descent.*

To illustrate in one dimension, consider the simple parabolic bowl

$$E(x) = \frac{1}{2}x^2.$$

For a given x, "downhill" is equivalent to making a move in the direction

$$-\frac{dE(x)}{dx} = -x$$

In other words, if x is positive and we are on the right hand of the bowl, we wish to take a step to the left. If x is on the left, the step should be taken to the right. Both cases take us closer to the minimum at $x = 0$. Furthermore, the size of the step is proportional to x. If, for example, we are far away from the minimum, a large step is taken. The steepest descent procedure is written as

$$\begin{aligned} x \Leftarrow \quad & x - \eta\tfrac{dE(x)}{dx} \\ & = (1-\eta)x. \end{aligned}$$

The new value of x is replaced by its old value multiplied by $(1-\eta)$. The parameter η is referred to as the *step size.* With an initialization of x_0, the solution of this difference equation after the pth step is

$$x_p = (1-\eta)^p x_0$$

This clearly converges to the minimum at $x = 0$ for all $|1-\eta| < 1$. Typically, the convergence of steepest decent cannot be analyzed in closed form as it was for this simple problem.

3.2.2.1. THE ERROR BACKPROPAGATION ALGORITHM

With reference to Figure 3.4, we will apply steepest descent to adjust the weight $w_{jk}(\ell)$. This corresponds to the interconnect between the kth neuron in layer $\ell - 1$ and the jth neuron in layer ℓ. As a function of $w_{jk}(\ell)$, the direction of steepest descent is simply

$$-\frac{\partial E}{\partial w_{jk}(\ell)}$$

where E is the error at the output given by Equation 3.6. Thus, if a step is taken in this direction, the new weight is

$$w_{jk}(\ell) \Leftarrow w_{jk}(\ell) - \eta\frac{\partial E}{\partial w_{jk}(\ell)} \tag{3.7}$$

where η is the step size. Making this adjustment to all of the weights nudges the neural network toward a more accurate response to the applied training data input and target.

To evaluate the steepest descent direction in Equation 3.7, the partial fraction is expanded

$$\frac{\partial E}{\partial w_{jk}(\ell)} = \frac{\partial E}{\partial s_j(\ell)}\frac{\partial s_j(\ell)}{\partial \mathrm{sum}_j(\ell)}\frac{\partial \mathrm{sum}_j(\ell)}{\partial w_{jk}(\ell)} \tag{3.8}$$

For later reference, define the error derivative as

$$\delta_j(\ell) = \frac{\partial E}{\partial s_j(\ell).} \tag{3.9}$$

The remaining two terms in Equation 3.8 are now analyzed. If the sigmoid in Equation 3.2 is used, then, from Equation 3.3,

$$\frac{\partial s_j(\ell)}{\partial \mathrm{sum}_j(\ell)} = s_j(\ell)\left[1 - s_j(\ell)\right] \tag{3.10}$$

Lastly, from Equation 3.1,

$$\begin{aligned} \frac{\partial \mathrm{sum}_j(\ell)}{\partial w_{jk}(\ell)} &= \frac{\partial}{\partial w_{jk}(\ell)} \sum_{p\in \text{ layer } (\ell-1)} w_{jp}(\ell)s_p(\ell-1) \\ &= \frac{\partial}{\partial w_{jk}(\ell)}[w_{j1}(\ell)s_1(\ell-1) + w_{j2}(\ell)s_2(\ell-1) \\ &\quad + \cdots + w_{jk}(\ell)s_k(\ell-1) + \cdots \\ &\quad + w_{jK_{\ell-1}}(\ell)s_{K_{\ell-1}}(\ell-2) \\ &= s_k(\ell-1) \end{aligned} \tag{3.11}$$

where K_ℓ is the number of neurons in layer ℓ. Substituting Equations 3.9-11into Equation 3.8 gives

$$\frac{\partial E}{\partial w_{jk}(\ell)} = \delta_j(\ell)s_j(\ell)\left[1 - s_j(\ell)\right]s_k(\ell-1) \tag{3.12}$$

All of the state terms, $s_j(\ell)$, in Equation 3.12 were computed when the input was propagated to the output and are available for updating the weight. The only term that remains a mystery is $\delta_j(\ell)$. For the output layer of neurons, this term can be computed directly. For other layers, it is determined by error backpropagation.

For the output layer, $\vec{s}(L) = \vec{o}$ = the output vector. Thus, from Equations 3.9 and 3.6,

$$\begin{aligned}\delta_j(1) &= \frac{\partial E}{\partial o_j} \\ &= \frac{\partial}{\partial o_j}\left[\frac{1}{2}\sum_{m=1}^{M}(o_m - t_m)^2\right] \\ &= o_j - t_j \end{aligned} \tag{3.13}$$

For weights not connected to the output, the value of the δ_j's can be computed from the δ_j's in the layer above them. The value of $\delta_j(L-1)$ can be evaluated given the values of the $\delta_j(L)$, the values of $\delta_j(L-2)$ can then be evaluated with knowledge of the $\delta_j(L-1)$'s etc. The procedure is then to compute the values at the output and work backwards towards the input or, in other words, to *backpropagate* the error from the output to the input. Doing so allows computation of all of the $\delta_j(\ell)$'s in the neural network and thereby allow each weight in the neural networks to be updated in accordance to steepest descent given by Equation 3.7.

In order to see how the error is backpropagated, expand Equation 3.9 into a partial fraction expansion

$$\begin{aligned}\delta_j(\ell) &= \frac{\partial E}{\partial s_j(\ell).} \\ &= \sum_{k\in \text{ the } (\ell+1)\text{st layer}} \frac{\partial E}{\partial s_k(\ell+1)} \\ &\quad \cdot \frac{\partial s_k(\ell+1)}{\partial \text{sum}_k(\ell+1)} \frac{\partial \text{sum}_k(\ell+1)}{\partial s_j(\ell)} \end{aligned} \tag{3.14}$$

As before, each of these terms can be evaluated individually. First, from Equation 3.9,

$$\frac{\partial E}{\partial s_k(\ell+1)} = \delta_k(\ell+1). \tag{3.15}$$

Assuming the sigmoid nonlinearity of Equation 3.2, the second term can be written, using Equation 3.9, as

$$\frac{\partial s_k(\ell+1)}{\partial \text{sum}_k(\ell+1)} = s_k(\ell+1)\left[1 - s_k(\ell+1)\right]. \tag{3.16}$$

Lastly

$$\begin{aligned}\frac{\text{sum}_k(\ell+1)}{\partial s_j(\ell)} &= \frac{\partial \text{sum}_k(\ell+1)}{\partial s_j(\ell)} \\ &= \frac{\partial}{\partial s_j(\ell)} \sum_{k\in \text{ the } \ell\text{th layer}} w_{jk}(\ell+1)s_k(\ell) \\ &= w_{jk}(\ell) \end{aligned} \tag{3.17}$$

Using Equations 3.15-17, Equation 14 can be written as

$$\begin{aligned}\delta_j(\ell) &= \sum_{k\in \text{ the } (\ell+1)\text{st layer}} \delta_k(\ell+1) \\ &\quad \cdot s_k(\ell+1)\left[1 - s_k(\ell+1)\right] w_{jk}(\ell) \end{aligned} \tag{3.18}$$

The δ's for each neuron can therefore be evaluated from the δ's of the neurons in the row above. The δ's for the top (Lth or output) row is simply the difference between the actual and desired (target) output.

3.2.2.2. BACKPROPAGATION SUMMARY

A summary of the steps for error backpropagation training of a layered perceptron is below. In practice, there are additional items that must be taken into account in the training. These are listed in the next section. Incorporation of bias and in the architecture and momentum in the training are of particular importance.

1. Set $n = 1$.

2. **The states of the neurons are determined by a feedforward operation on the input.** For an input of $\vec{i} = \vec{i_n}$, evaluate the states, $s_j(\ell)$, of all of the neurons in each layer of the neural network according to the formula

$$s_j(\ell) = \sigma\left(\sum_{k=1}^{K_{\ell-1}} w_{jk}(\ell)s_k(\ell-1)\right) \tag{3.19}$$

where

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

The recursion is initiated using $s_j(0) = i_j$. The states of the the final layer are the output (*i.e.* $\vec{o} = \vec{s}_j(L)$.

Example: At the top of Figure 3.5, the state of the second neuron in the second layer, $s_2(2)$, is determined by the states of the neurons in layer one, specifically $s_1(1)$, $s_2(1)$ and $s_3(1)$. The value is computed using Equation 3.18.

3. **Backpropagate the error** between the actual and desired neural network outputs to assign $\delta_j(\ell)$'s to each neuron in each layer. The formula is

$$\delta_j(\ell) = \begin{cases} o_j - t_j & \ell = L \\ \sum_{k=1}^{K_{\ell+1}} \delta_k(\ell+1)s_k(\ell+1) & \\ \cdot\left[1 - s_k(\ell+1)\right] w_{jk}(\ell) & ;\ 1 \le \ell \le L-1 \end{cases} \tag{3.20}$$

where t_j is the jth component of the target vector $\vec{t} = \vec{t_n}$, the output vector is $\vec{o} = \vec{s}_L$ and K_ℓ is the number of neurons in the ℓth layer.

Example: In the center of Figure3.5, the error derivative of the third neuron in layer one, $\delta_3(1)$,

is evaluated as a function of the error derivatives above it, specifically $\delta_1(2)$ and $\delta_2(2)$. Equation 3.19 is used for this.

4. **Update weights using steepest descent.** From the previous two steps, each neuron is now assigned a state, $s_{jk}(\ell)$ and a error derivative, $\delta_{jk}(\ell)$. The weights are updated using

$$\begin{aligned} w_{jk}(\ell) &\Leftarrow w_{jk}(\ell) - \eta \frac{\partial E}{\partial w_{jk}(\ell)} \\ &= w_{jk}(\ell) + \Delta w_{jk}(\ell) \end{aligned} \quad (3.21)$$

where

$$\Delta w_{jk}(\ell) = \eta \delta_j(\ell) s_j(\ell) [1 - s_j(\ell)] s_k(\ell - 1) \quad (3.22)$$

Example: Weight $w_{31}(1)$ is updated in the bottom figure in Figure3.5 using the state and error derivative of the top neuron ($s_3(2)$ and $\delta_3(2)$) and the state of the lower neuron ($s_1(0) = i_1$). The interconnect joins these two neurons.

5. If $n < N$, repeat the procedure from step 2 for the training pair $\vec{i}_{n+1}$ and $\vec{t}_{n+1}$.

6. The neural network has been subjected to a cycle of training data, or an *epoch*. If the error,

$$E = \frac{1}{2} \sum_{m=1}^{M} (o_m - t_m)^2$$

is sufficiently small, stop. Otherwise, go to step 1 and go through another epoch.

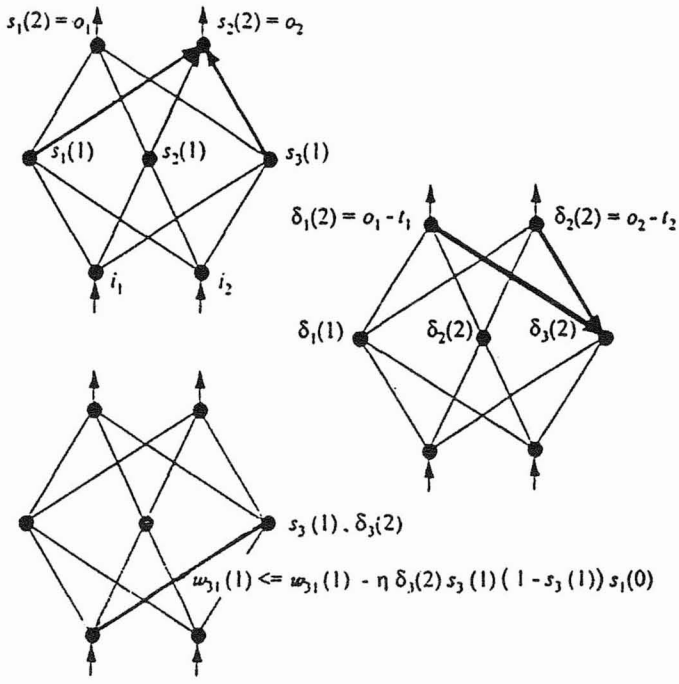


Figure 3.5 An example of training a layered perceptron using error backpropagation. (Top) In the feedforward step, the states of all neurons are determined directly from the layer of neurons below them. (Middle) The error between the actual and target values is backpropagated to assign each neuron an error derivative denoted by δ. The error derivatives are determined directly from the error derivatives of the neurons in the layer above. (Bottom) Each neuron now has a state and as error derivative. The weight connecting two neurons is updated using steepest descent. Only the states and error derivatives of the connecting neurons are needed to do this.

3.3. NEURAL SMITHING

There are numerous variations in training a layered perceptron neural network.

- **Momentum.** Error backpropagation as described many times does not work. Weights can be adjusted in such a helter skelter manner that no convergence occurs. A technique to alleviate this adds *momentum* to the weight update. Momentum requires the weight to change in a direction akin to its previous change.

  To incorporate momentum, an additional parameter must be added to the weight increment in Equation 3.21. Redefine $\Delta w_{jk}(\ell)$ as $\Delta w_{jk}^m(\ell)$ where the new parameter m indexes the number of epochs, or passes through the training data. To include momentum, Equation 3.21 is rewritten as

$$\begin{aligned} \Delta w_{jk}^m(\ell) &= \eta \delta_j(\ell) s_j(\ell) [1 - s_j(\ell)] s_k(\ell - 1) \\ &\quad -\alpha \Delta w_{jk}(\ell)^{m-1} \end{aligned}$$

  where α, the momentum, parameterizes the weight of the previous weight increment on the current one.

- **Bias.** One of the inputs is set to one to provide a bias to the neural network. In other words, set $i_1 = 1$ for all inputs. Doing so allows greater diversity in the ability of the neural network to learn. To illustrate, consider the case where $\tanh(\cdot)$ is used in lieu of the sigmoid in Equation 3.2. (Indeed, $\tanh(x) = 2(\sigma(x) - 1)$.) Since $\tanh(0) = 0$, an input identically equal to zero will result in an output identically equal to zero. There exists no freedom to make any other assignment. Addition of a bias term on the input removes this restriction. Bias nodes are sometimes added to hidden layers also.

- **Shuffling.** The ordering of the data can have an unwanted effect on the training of the neural network. Some neural smiths randomize the ordering of the training data for each epoch.

- **Batch Training.** This process is used more commonly than shuffling. An entire epoch is presented

to the neural network prior to updating the weights. With reference to Equation 3.5, the total error for an epoch is

$$E = \sum_{n=1}^{N} E_n labelepp \qquad (3.23)$$

where N is the cardinality of the training data input-output pairs. This error, rather than the error from a single input-output training data pair, is used to update the weights of the neural network.

- **Cross Validation.** There is a difference between *learning* and *memorization*. With memorization, data previously encountered can be properly categorized. Data not previously seen may or may not be properly categorized. If memorization is desired, a table look-up might be preferable to a neural network.

  If a neural network's architecture is improperly chosen by, for example, choosing too many hidden neurons, then the neural network may memorize. Techniques, such as hidden neuron *pruning* can be used to remedy this.

  How do we determine whether a neural network is properly trained? The neural network will properly respond to data which it has not seen before. Subjecting the neural network to such data is referred to as *cross validation*.

- **Hidden Layer Choice.** A single hidden layer is sufficient to make any input-ouput mapping. A single hidden layer, though, may not be best. Layered perceptrons with a very large number of hidden layers tend to train poorly.

- **Pruning.** A layered perceptron with too many hidden neurons will tend to memorize the training data. Too few hidden neurons may not allow sufficient flexibility to generalize. One method to determine the proper number of hidden neurons to begin with a large number of neurons and *prune* hidden neurons until desired performance is obtained. One method prunes as long as the cross validation error decreases.

- **Sparse Data and Training with Jitter.** When the training data is sparse or to improve generalization, the input data can be corrupted with noise. The idea is that each point claims more volume in the training space.

- **Alternate Training Techniques** Training a neural network is a classic optimization problem. A search is made over weight space in order to minimize the overall error for a given training data set. Error backpropagation has the advantage that all of the training can be done within the neural network architecture. If this is not a concern, then any of a number of optimization techniques can be used. Some of the techniques that have been suggested are listed below.

  1. Conjugate gradient descent.
  2. Random optimization.
  3. Genetic algorithms.

- **The Curse of Dimensionality.** The number of inputs to a layered perceptron should be as few as possible. This is a guideline not only for neural networks, but for any classification procedure.

  Consider, for example, the circle in a square pictured in Figure 3.6. In order to learn to distinguish between points in the circle and points without, a large number of training points are required. The neural network would have two inputs corresponding to the coordinates of each point. The single output of the neural network would be trained to one for points within the circle and zero without. How many points are required in order to see the classification boundary is circular? One hundred points, shown at the top of Figure 3.6. Consider, then, the same problem extended to three dimensions illustrated at the bottom of Figure 3.6. An additional input, i_3, has been added. It has no effect on the classification. The problem is to classify within or without the cylinder. Even though the problem is basically the same as before, the required number of training data is much higher. In order to have the same resolution as in the two dimensional case, one hundred slices of one hundred samples is required. The number of samples is therefore 100,000 rather than 100. Generalizing, the number of samples increases as S^K where S is the number of samples in one dimension and K is the number of inputs. This simple example illustrates the need to reduce the input dimensionality as much as possible without destroying the information theoretic content.

3.4. VARIATIONS

There exist an extensive number of variations of the layered perceptron.

- **Radial Basis Function Neural Networks.** Instead of the sigmoid, a gaussian is used as the neural nonlinearity. The dispersion and centroid of these gaussians are tuned in the training process.

- **Recurrent Neural Networks.** These nets have

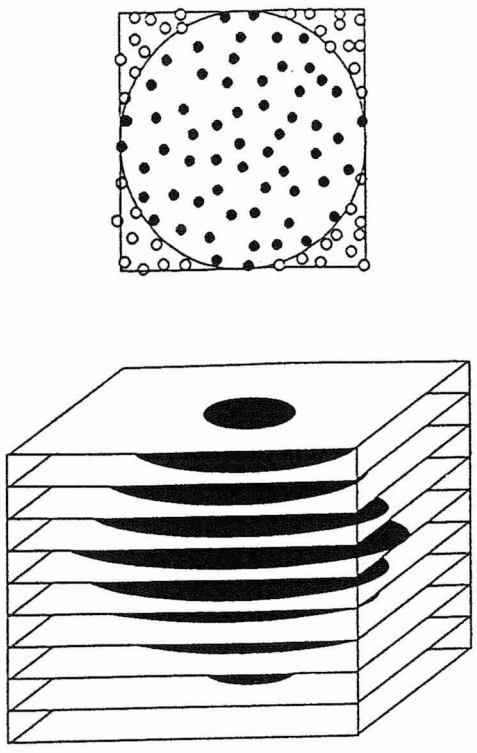


Figure 3.6 Illustration of the curse of dimensionality. The top classification problem is characterized by 100 samples. In order to generate the same resolution in the bottom three dimensional problem, 100,000 samples are required.

feedback interconnects. The feedforward neural network only has the ability to provide memoryless input-output mappings. Recurrent neural networks can make use of past events to make decisions. As in any system with feedback, stability can be a problem. Chaos can be observed in such neural networks.

- **Cascade Correlation.** The neural network is trained to a certain accuracy. Additional neurons are added, if needed, to improve performance.

3.5.ADDITIONAL READING

There exist numerous quality tutorials and texts on neural networks. The books by Hassoun, Haykin and Zurada are especially good. Shorter tutorials on neural networks are given by Hush & Horne and by Simpson.

The three dominant jounals in the field are *The IEEE Transactions on Neural Networks, Neural Computation* and *Neural Networks.*

BIBLIOGRAPHY

1. P. Arabshahi, J.J. Choi, R.J. Marks II and T.P. Caudell, "Fuzzy Control of Backpropagation," *Proc. First IEEE Int. Conf. Fuzzy Systems,* San Diego, CA, 1992 (IEEE Press).

2. M.H. Hassoun, **Fundamentals of Artificial Neural Networks**, MIT Press, 1995.

3. Simon Haykin, **Neural Networks: A Comprehensive Foundation**, (Macmillan/IEEE Press, 1994.)

4. D.R. Hush and B.G. Horne, "Progress in Supervised Neural Networks", *IEEE Signal Processing Magazine*, pp 8-39, January 1993.

5. R. Reed, "Pruning Algorithms - A Survey", *IEEE Transactions on Neural Networks*, vol. 4, #5, pp.740-747.

6. R. Reed and R.J. Marks II, "Genetic Algorithms and Neural Networks: An Introduction", *Northcon/92 Conference Record*, (Western Periodicals Co., Ventura, CA), Seattle WA, October 19-21, 1992, pp.293-301 - invited paper.

7. Patrick K. Simpson, *Foundations of Neural Networks*, in **Artificial Neural Networks: Paradigms, Applications and Hardware Implementations**, Sanchez-Sinencio & Lau, Editors, IEEE Press, 1992.

8. J.M. Zurada, **Introduction to Artificial Neural Systems**, West Publishing Company, St. Paul, MN, 1992.

9. J. Zurada, R.J. Marks II and C. Robinson, **Computational Intelligence: Imitating Life**, (IEEE Press, Piscataway, 1994).