Principe, J.C. “Artificial Neural Networks”
The Electrical Engineering Handbook
Ed. Richard C. Dorf
Boca Raton: CRC Press LLC, 2000

20

Artificial Neural Networks

20.1 Definitions and Scope
Introduction • Definitions and Style of Computation • ANN Types and Applications
20.2 Multilayer Perceptrons
Function of Each PE • How to Train MLPs • Applying Back-Propagation in Practice • *A Posteriori* Probabilities
20.3 Radial Basis Function Networks
20.4 Time Lagged Networks
Memory Structures • Training-Focused TLN Architectures
20.5 Hebbian Learning and Principal Component Analysis Networks
Hebbian Learning • Principal Component Analysis • Associative Memories
20.6 Competitive Learning and Kohonen Networks

Jose C. Principe
University of Florida

20.1 Definitions and Scope

Introduction

Artificial neural networks (ANN) are among the newest signal-processing technologies in the engineer's toolbox. The field is highly interdisciplinary, but our approach will restrict the view to the engineering perspective. In engineering, neural networks serve two important functions: as pattern classifiers and as nonlinear adaptive filters. We will provide a brief overview of the theory, learning rules, and applications of the most important neural network models.

Definitions and Style of Computation

An ANN is an adaptive, most often nonlinear system that learns to perform a function (an input/output map) from data. Adaptive means that the system parameters are changed during operation, normally called the *training phase*. After the training phase the ANN parameters are fixed and the system is deployed to solve the problem at hand (the *testing phase*). The ANN is built with a systematic step-by-step procedure to optimize a performance criterion or to follow some implicit internal constraint, which is commonly referred to as the *learning rule*. The input/output training data are fundamental in neural network technology, because they convey the necessary information to "discover" the optimal operating point. The nonlinear nature of the neural network processing elements (PEs) provides the system with lots of flexibility to achieve practically any desired input/output map, i.e., some ANNs are *universal mappers*.

There is a style in neural computation that is worth describing (Fig. 20.1). An input is presented to the network and a corresponding desired or target response set at the output (when this is the case the training is called *supervised*). An error is composed from the difference between the desired response and the system

© 2000 by CRC Press LLC

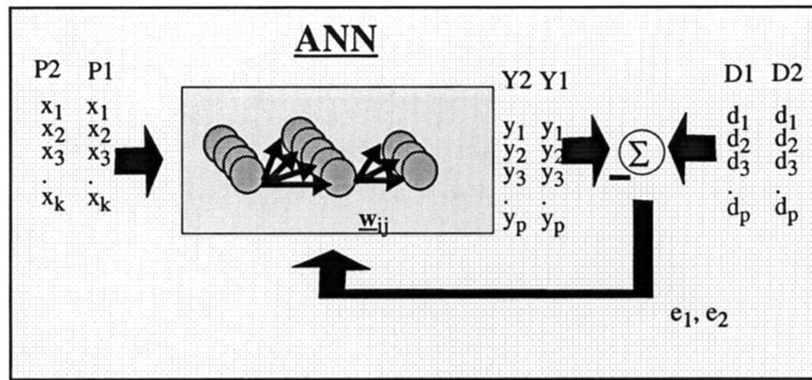

FIGURE 20.1 The style of neural computation.

output. This error information is fed back to the system and adjusts the system parameters in a systematic fashion (the learning rule). The process is repeated until the performance is acceptable. It is clear from this description that the performance hinges heavily on the data. If one does not have data that cover a significant portion of the operating conditions or if they are noisy, then neural network technology is probably not the right solution. On the other hand, if there is plenty of data and the problem is poorly understood to derive an approximate model, then neural network technology is a good choice.

This operating procedure should be contrasted with the traditional engineering design, made of exhaustive subsystem specifications and intercommunication protocols. In ANNs, the designer chooses the network topology, the performance function, the learning rule, and the criterion to stop the training phase, but the system automatically adjusts the parameters. So, it is difficult to bring *a priori* information into the design, and when the system does not work properly it is also hard to incrementally refine the solution. But ANN-based solutions are extremely efficient in terms of development time and resources, and in many difficult problems ANNs provide performance that is difficult to match with other technologies. Denker 10 years ago said that "ANNs are the second best way to implement a solution" motivated by the simplicity of their design and because of their universality, only shadowed by the traditional design obtained by studying the physics of the problem. At present, ANNs are emerging as the technology of choice for many applications, such as pattern recognition, prediction, system identification, and control.

ANN Types and Applications

It is always risky to establish a taxonomy of a technology, but our motivation is one of providing a quick overview of the application areas and the most popular topologies and learning paradigms.

Application	Topology	Supervised Learning	Unsupervised Learning
Association	Hopfield [Zurada, 1992; Haykin, 1994]	—	Hebbian [Zurada, 1992; Haykin, 1994; Kung, 1993]
	Multilayer perceptron [Zurada, 1992; Haykin, 1994; Bishop, 1995]	Back-propagation [Zurada, 1992; Haykin, 1994; Bishop, 1995]	—
	Linear associative mem. [Zurada, 1992; Haykin, 1994]	—	Hebbian
Pattern recognition	Multilayer perceptron [Zurada, 1992; Haykin, 1994; Bishop, 1995]	Back-propagation	—
	Radial basis functions [Zurada, 1992; Bishop, 1995]	Least mean square	*k*-means [Bishop, 1995]
Feature extraction	Competitive [Zurada, 1992; Haykin, 1994]	—	Competitive
	Kohonen [Zurada, 1992; Haykin, 1994]	—	Kohonen
	Multilayer perceptron [Kung, 1993]	Back-propagation	—
	Principal comp. anal. [Zurada, 1992; Kung, 1993]	—	Oja's [Zurada, 1992; Kung, 1993]
Prediction, system ID	Time-lagged networks [Zurada, 1992; Kung, 1993; de Vries and Principe, 1992]	Back-propagation through time [Zurada, 1992]	—
	Fully recurrent nets [Zurada, 1992]		

© 2000 by CRC Press LLC

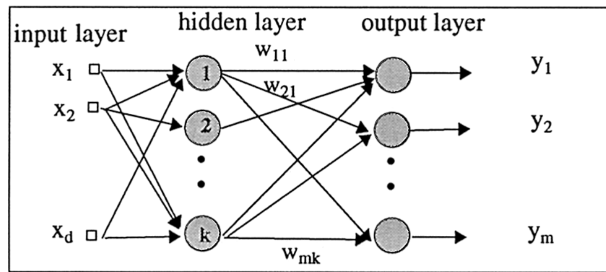

FIGURE 20.2 MLP with one hidden layer (d-k-m).

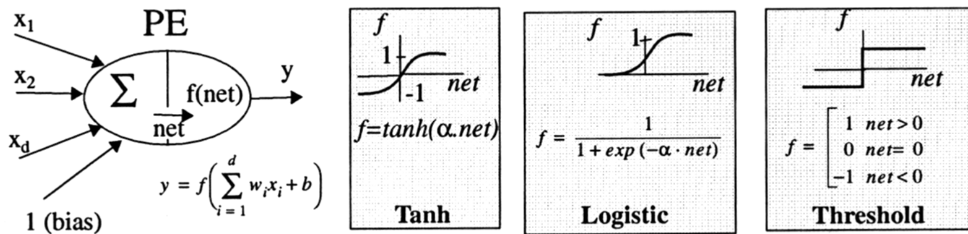

FIGURE 20.3 A PE and the most common nonlinearities.

It is clear that *multilayer perceptrons* (MLPs), the *back-propagation algorithm* and its extensions — time-lagged networks (TLN) and back-propagation through time (BPTT), respectively — hold a prominent position in ANN technology. It is therefore only natural to spend most of our overview presenting the theory and tools of back-propagation learning. It is also important to notice that *Hebbian learning* (and its extension, the Oja rule) is also a very useful (and biologically plausible) learning mechanism. It is an *unsupervised learning* method since there is no need to specify the desired or target response to the ANN.

20.2 Multilayer Perceptrons

Multilayer perceptrons are a layered arrangement of nonlinear PEs as shown in Fig. 20.2. The layer that receives the input is called the *input layer*, and the layer that produces the output is the *output layer*. The layers that do not have direct access to the external world are called *hidden layers*. A layered network with just the input and output layers is called the *perceptron*. Each connection between PEs is weighted by a scalar, w_i, called a *weight*, which is adapted during learning.

The PEs in the MLP are composed of an adder followed by a smooth saturating nonlinearity of the sigmoid type (Fig. 20.3). The most common saturating nonlinearities are the logistic function and the hyperbolic tangent. The threshold is used in other nets. The importance of the MLP is that it is a universal mapper (implements arbitrary input/output maps) when the topology has at least two hidden layers and sufficient number of PEs [Haykin, 1994]. Even MLPs with a single hidden layer are able to approximate continuous input/output maps. This means that rarely we will need to choose topologies with more than two hidden layers. But these are existence proofs, so the issue that we must solve as engineers is to choose how many layers and how many PEs in each layer are required to produce good results.

Many problems in engineering can be thought of in terms of a transformation of an input space, containing the input, to an output space where the desired response exists. For instance, dividing data into classes can be thought of as transforming the input into 0 and 1 responses that will code the classes [Bishop, 1995]. Likewise, identification of an unknown system can also be framed as a mapping (function approximation) from the input to the system output [Kung, 1993]. The MLP is highly recommended for these applications.

© 2000 by CRC Press LLC

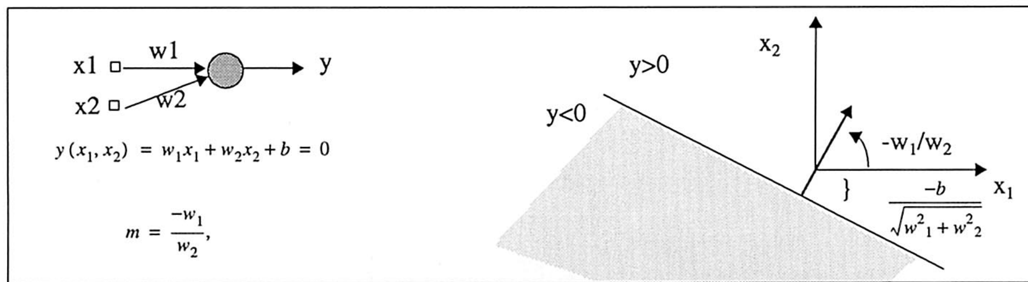

FIGURE 20.4 A two-input PE and its separation surface.

Function of Each PE

Let us study briefly the function of a single PE with two inputs [Zurada, 1992]. If the nonlinearity is the threshold nonlinearity we can immediately see that the output is simply 1 and –1. The surface that divides these subspaces is called a *separation surface,* and in this case it is a line of equation

$$y(w_1, w_2) = w_1 x_1 + w_2 x_2 + b = 0 \tag{20.1}$$

i.e., the PE weights and the bias control the orientation and position of the separation line, respectively (Fig. 20.4). In many dimensions the separation surface becomes an hyperplane of dimension one less than the dimensionality of the input space. So, each PE creates a dichotomy in the input space. For smooth nonlinearities the separation surface is not crisp; it becomes fuzzy but the same principles apply. In this case, the size of the weights controls the width of the fuzzy boundary (larger weights shrink the fuzzy boundary).

The perceptron input/output map is built from a juxtaposition of linear separation surfaces, so the perceptron gives zero classification error only for *linearly separable classes* (i.e., classes that can be exactly classified by hyperplanes).

When one adds one layer to the perceptron creating a one hidden layer MLP, the type of separation surfaces changes drastically. It can be shown that this learning machine is able to create "bumps" in the input space, i.e., an area of high response surrounded by low responses [Zurada, 1992]. The function of each PE is always the same, no matter if the PE is part of a perceptron or an MLP. However, notice that the output layer in the MLP works with the result of hidden layer activations, creating an embedding of functions and producing more complex separation surfaces. The one-hidden-layer MLP is able to produce *nonlinear separation surfaces.*

If one adds an extra layer (i.e., two hidden layers), the learning machine now can combine at will bumps, which can be interpreted as a *universal mapper,* since there is evidence that any function can be approximated by localized bumps. One important aspect to remember is that changing a single weight in the MLP can drastically change the location of the separation surfaces; i.e., the MLP achieves the input/output map through the interplay of all its weights.

How to Train MLPs

One fundamental issue is how to adapt the weights w_i of the MLP to achieve a given input/output map. The core ideas have been around for many years in optimization, and they are extensions of well-known engineering principles, such as the *least mean square (LMS) algorithm* of adaptive filtering [Haykin, 1994]. Let us review the theory here. Assume that we have a linear PE (f(net) = net) and that one wants to adapt the weights as to minimize the square difference between the desired signal and the PE response (Fig. 20.5).

This problem has an analytical solution known as the *least squares* [Haykin, 1994]. The optimal weights are obtained as the product of the inverse of the input autocorrelation function (R^{-1}) and the cross-correlation vector (**P**) between the input and the desired response. The analytical solution is equivalent to a search for the minimum of the quadratic performance surface $J(w_i)$ using gradient descent, where the weights at each iteration k are adjusted by

© 2000 by CRC Press LLC

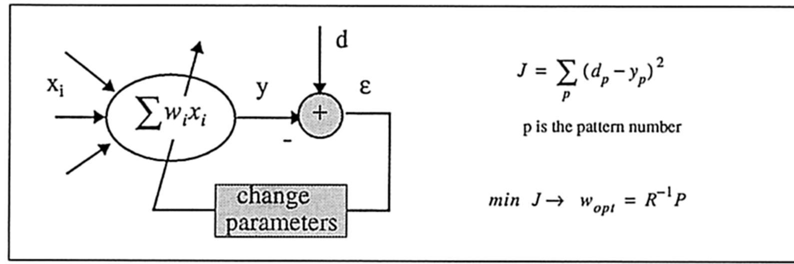

FIGURE 20.5 Computing analytically optimal weights for the linear PE.

$$w_i(k+1) = w_i(k) - \eta \nabla J_i(k) \qquad \nabla J_i = \frac{\partial J}{\partial w_i} \tag{20.2}$$

where η is a small constant called the *step size*, and $\nabla J(k)$ is the gradient of the performance surface at iteration *k*. Bernard Widrow in the late 1960s proposed a very efficient estimate to compute the gradient at each iteration

$$\nabla J_i(k) = \frac{\partial}{\partial w_i} J(k) \qquad \sim \frac{1}{2} \frac{\partial}{\partial w_i} \left(\varepsilon^2(k) \right) = -\varepsilon(k)\, x_i(k) \tag{20.3}$$

which when substituted into Eq. (20.2) produces the so-called *LMS algorithm*. He showed that the LMS converged to the analytic solution provided the step size η is small enough. Since it is a steepest descent procedure, the largest step size is limited by the inverse of the largest eigenvalue of the input autocorrelation matrix. The larger the step size (below this limit), the faster is the convergence, but the final values will "rattle" around the optimal value in a basin that has a radius proportional to the step size. Hence, there is a fundamental trade-off between speed of convergence and accuracy in the final weight values. One great appeal of the LMS algorithm is that it is very efficient (just one multiplication per weight) and requires only local quantities to be computed.

The LMS algorithm can be framed as a computation of partial derivatives of the cost with respect to the unknowns, i.e., the weight values. In fact, with the chainrule one writes

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial w_i} = \frac{\partial}{\partial y} \left(\Sigma (d - y)^2 \right) \frac{\partial}{\partial w_i} \left(\Sigma\, w_i\, x_i \right) = -\varepsilon x_i \tag{20.4}$$

we obtain the LMS algorithm for the linear PE. What happens if the PE is nonlinear? If the nonlinearity is differentiable (smooth), we still can apply the same method, because of the chain rule, which prescribes that (Fig. 20.6)

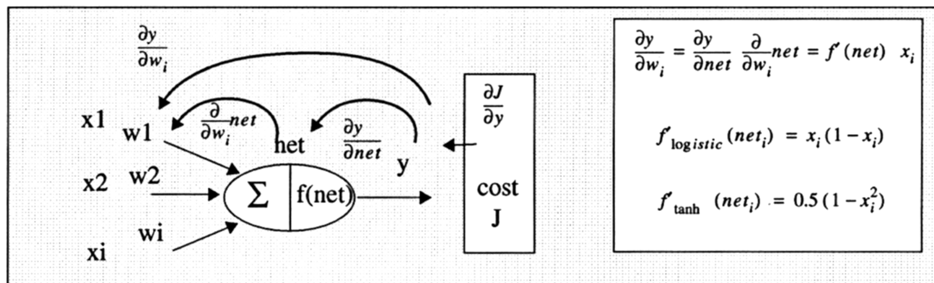

FIGURE 20.6 How to extend LMS to nonlinear PEs with the chain rule.

© 2000 by CRC Press LLC

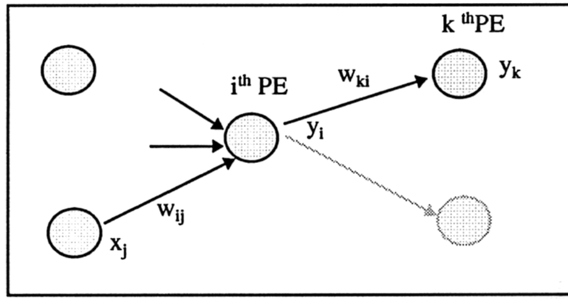

FIGURE 20.7 How to adapt the weights connected to *i*th PE.

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial \text{net}}\frac{\partial}{\partial w_i}net = -(d - y)f'(\text{net})\,x_i = -\varepsilon f'(\text{net})\,x_i \tag{20.5}$$

where $f'(\text{net})$ is the derivative of the nonlinearity computed at the operating point. Equation (20.5) is known as the *delta rule*, and it will train the perceptron [Haykin, 1994]. Note that throughout the derivation we skipped the pattern index p for simplicity, but this rule is applied for each input pattern. However, the delta rule cannot train MLPs since it requires the knowledge of the error signal at each PE.

The principle of the ordered derivatives can be extended to multilayer networks, provided we organize the computations in flows of activation and error propagation. The principle is very easy to understand, but a little complex to formulate in equation form [Haykin, 1994].

Suppose that we want to adapt the weights connected to a hidden layer PE, the *i*th PE (Fig. 20.7). One can decompose the computation of the partial derivative of the cost with respect to the weight w_{ij} as

$$\frac{\partial J}{\partial w_{ij}} = \underbrace{\frac{\partial J}{\partial y_i}}_{1}\underbrace{\frac{\partial y_i}{\partial \text{net}_i}\frac{\partial}{\partial w_{ij}}\text{net}_i}_{2} \tag{20.6}$$

i.e., the partial derivative with respect to the weight is the product of the partial derivative with respect to the PE state — part 1 in Eq. (20.6) — times the partial derivative of the local activation to the weights — part 2 in Eq. (20.6). This last quantity is exactly the same as for the nonlinear PE ($f'(\text{net}_i)x_j$), so the big issue is the computation of $\frac{\partial J}{\partial y}$. For an output PE, $\frac{\partial J}{\partial y}$ becomes the injected error ε in Eq. (20.4). For the hidden *i*th PE $\frac{\partial J}{\partial y}$ is evaluated by summing all the errors that reach the PE from the top layer through the topology when the injected errors ε_k are clamped at the top layer, or in an equation

$$\frac{\partial J}{\partial y_i} = \left(\sum_k \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial \text{net}_k}\frac{\partial}{\partial y_i}\text{net}_k\right) = \sum_k \varepsilon_k f'(\text{net}_k) w_{ki} \tag{20.7}$$

Substituting back in Eq. (20.6) we finally get

$$\frac{\partial J}{\partial w_{ij}} = -\underbrace{x_j}_{1}\underbrace{f'(\text{net}_i)\left(\sum_k \varepsilon_k\, f'(\text{net}_k)\, w_{ki}\right)}_{2} \tag{20.8}$$

© 2000 by CRC Press LLC

This equation embodies the *back-propagation training algorithm* [Haykin, 1994; Bishop, 1995]. It can be rewritten as the product of a local activation (part 1) and a local error (part 2), exactly as the LMS and the delta rules. But now the local error is a composition of errors that flow through the topology, which becomes equivalent to the existence of a desired response at the PE.

There is an intrinsic flow in the implementation of the back-propagation algorithm: first, inputs are applied to the net and activations computed everywhere to yield the output activation. Second, the external errors are computed by subtracting the net output from the desired response. Third, these external errors are utilized in Eq. (20.8) to compute the local errors for the layer immediately preceding the output layer, and the computations chained up to the input layer. Once all the local errors are available, Eq. (20.2) can be used to update every weight. These three steps are then repeated for other training patterns until the error is acceptable.

Step three is equivalent to injecting the external errors in the *dual topology* and back-propagating them up to the input layer [Haykin, 1994]. The dual topology is obtained from the original one by reversing data flow and substituting summing junctions by splitting nodes and vice versa. The error at each PE of the dual topology is then multiplied by the activation of the original network to compute the weight updates. So, effectively the dual topology is being used to compute the local errors which makes the procedure highly efficient. This is the reason back-propagation trains a network of N weights with a number of multiplications proportional to N, ($O(N)$), instead of ($O(N^2)$) for previous methods of computing partial derivatives known in control theory. Using the dual topology to implement back-propagation is the best and most general method to program the algorithm in a digital computer.

Applying Back-Propagation in Practice

Now that we know an algorithm to train MLPs, let us see what are the practical issues to apply it. We will address the following aspects: size of training set vs. weights, search procedures, how to stop training, and how to set the topology for maximum generalization.

Size of Training Set

The size of the training set is very important for good performance. Remember that the ANN gets its information from the training set. If the training data do not cover the full range of operating conditions, the system may perform badly when deployed. Under no circumstances should the training set be less than the number of weights in the ANN. A good size of the training data is ten times the number of weights in the network, with the lower limit being set around three times the number of weights (these values should be taken as an indication, subject to experimentation for each case) [Haykin, 1994].

Search Procedures

Searching along the direction of the gradient is fine if the performance surface is quadratic. However, in ANNs rarely is this the case, because of the use of nonlinear PEs and topologies with several layers. So, gradient descent can be caught in local minima, which makes the search very slow in regions of small curvature. One efficient way to speed up the search in regions of small curvature and, at the same time, to stabilize it in narrow valleys is to include a momentum term in the weight adaptation

$$w_{ij}(n+1) = w_{ij}(n) + \eta\delta(n)x_j(n) + \alpha\left(w_{ij}(n) - w_{ij}(n-1)\right) \tag{20.9}$$

The value of momentum α should be set experimentally between 0.5 and 0.9. There are many more modifications to the conventional gradient search, such as adaptive step sizes, annealed noise, conjugate gradients, and second-order methods (using information contained in the Hessian matrix), but the simplicity and power of momentum learning is hard to beat [Haykin, 1994; Bishop, 1995].

How to Stop Training

The stop criterion is a fundamental aspect of training. The simple ideas of capping the number of iterations or of letting the system train until a predetermined error value are not recommended. The reason is that we want the ANN to perform well in the test set data; i.e., we would like the system to perform well in data it

© 2000 by CRC Press LLC

never saw before (good *generalization*) [Bishop, 1995]. The error in the training set tends to decrease with iteration when the ANN has enough degrees of freedom to represent the input/output map. However, the system may be remembering the training patterns (*overfitting*) instead of finding the underlying mapping rule. This is called *overtraining*. To avoid overtraining the performance in a *validation set*, i.e., a set of input data that the system never saw before, must be checked regularly during training (i.e., once every 50 passes over the training set). The training should be stopped when the performance in the validation set starts to increase, despite the fact that the performance in the training set continues to decrease. This method is called *cross validation*. The validation set should be 10% of the training set, and distinct from it.

Size of the Topology

The size of the topology should also be carefully selected. If the number of layers or the size of each layer is too small, the network does not have enough degrees of freedom to classify the data or to approximate the function, and the performance suffers.

On the other hand, if the size of the network is too large, performance may also suffer. This is the phenomenon of *overfitting* that we mentioned above. But one alternative way to control it is to reduce the size of the network. There are basically two procedures to set the size of the network: either one starts small and adds new PEs or one starts with a large network and prunes PEs [Haykin, 1994]. One quick way to prune the network is to impose a penalty term in the performance function — a *regularizing term* — such as limiting the slope of the input/output map [Bishop, 1995]. A regularization term that can be implemented locally is

$$w_{ij}(n+1) = w_{ij}(n)\left(1 - \frac{\lambda}{\left(1 + w_{ij}(n)\right)^2}\right) + \eta\delta_i(n)\,x_j(n) \tag{20.10}$$

where λ is the *weight decay* parameter and δ the local error. Weight decay tends to drive unimportant weights to zero.

A Posteriori Probabilities

We will finish the discussion of the MLP by noting that this topology when trained with the mean square error is able to estimate directly at its outputs *a posteriori* probabilities, i.e., the probability that a given input pattern belongs to a given class [Bishop, 1995]. This property is very useful because the MLP outputs can be interpreted as probabilities and operated as numbers. In order to guarantee this property, one has to make sure that each class is attributed to one output PE, that the topology is sufficiently large to represent the mapping, that the training has converged to the absolute minimum, and that the outputs are normalized between 0 and 1. The first requirements are met by good design, while the last can be easily enforced if the *softmax activation* is used as the output PE [Bishop, 1995],

$$y = \frac{\exp(\text{net})}{\sum_j \exp(\text{net}_j)} \tag{20.11}$$

20.3 Radial Basis Function Networks

The radial basis function (RBF) network constitutes another way of implementing arbitrary input/output mappings. The most significant difference between the MLP and RBF lies in the PE nonlinearity. While the PE in the MLP responds to the full input space, the PE in the RBF is local, normally a Gaussian kernel in the input space. Hence, it only responds to inputs that are close to its center; i.e., it has basically a *local response*.

© 2000 by CRC Press LLC

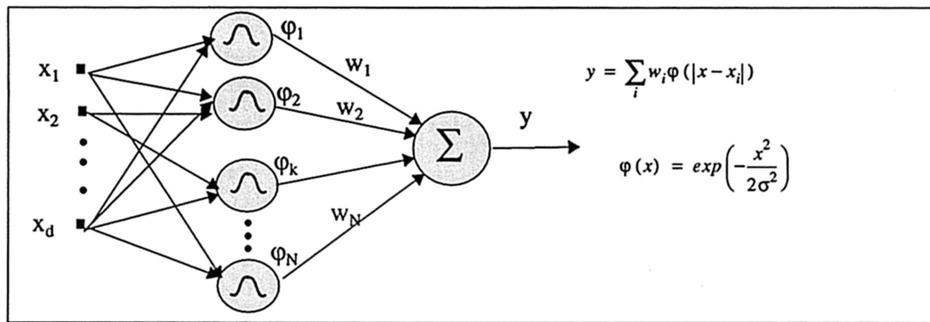

FIGURE 20.8 Radial Basis Function (RBF) network.

The RBF network is also a layered net with the hidden layer built from Gaussian kernels and a linear (or nonlinear) output layer (Fig. 20.8). Training of the RBF network is done normally in two stages [Haykin, 1994]: first, the centers x_i are adaptively placed in the input space using competitive learning or k means clustering [Bishop, 1995], which are unsupervised procedures. Competitive learning is explained later in the chapter. The variances of each Gaussian are chosen as a percentage (30 to 50%) to the distance to the nearest center. The goal is to cover adequately the input data distribution. Once the RBF is located, the second layer weights w_i are trained using the LMS procedure.

RBF networks are easy to work with, they train very fast, and they have shown good properties both for function approximation as classification. The problem is that they require lots of Gaussian kernels in high-dimensional spaces.

20.4 Time-Lagged Networks

The MLP is the most common neural network topology, but it can only handle instantaneous information, since the system has no memory and it is feedforward. In engineering, the processing of signals that exist in time requires systems with memory, i.e., linear filters. Another alternative to implement memory is to use feedback, which gives rise to *recurrent networks*. Fully recurrent networks are difficult to train and to stabilize, so it is preferable to develop topologies based on MLPs but where explicit subsystems to store the past information are included. These subsystems are called *short-term memory structures* [de Vries and Principe, 1992]. The combination of an MLP with short-term memory structures is called a *time-lagged network* (*TLN*). The memory structures can be eventually recurrent, but the feedback is local, so stability is still easy to guarantee. Here, we will cover just one TLN topology, called *focused*, where the memory is at the input layer. The most general TLN have memory added anywhere in the network, but they require other more-involved training strategies (BPTT [Haykin, 1994]). The interested reader is referred to de Vries and Principe [1992] for further details.

The function of a short-term memory in the focused TLN is to represent the past of the input signal, while the nonlinear PEs provide the mapping as in the MLP (Fig. 20.9).

Memory Structures

The simplest memory structure is built from a *tap delay line* (Fig. 20.10). The *memory by delays* is a single-input, multiple-output system that has no free parameters except its size K. The tap delay memory is the memory utilized in the *time-delay neural network* (TDNN) which has been utilized successfully in speech recognition and system identification [Kung, 1993].

A different mechanism for linear memory is the *feedback* (Fig. 20.11). Feedback allows the system to remember past events because of the exponential decay of the response. This memory has limited resolution because of the low pass required for long memories. But notice that unlike the memory by delay, memory by feedback provides the learning system with a free parameter μ that controls the length of the memory. Memory by feedback has been used in Elman and Jordan networks [Haykin, 1994].

© 2000 by CRC Press LLC

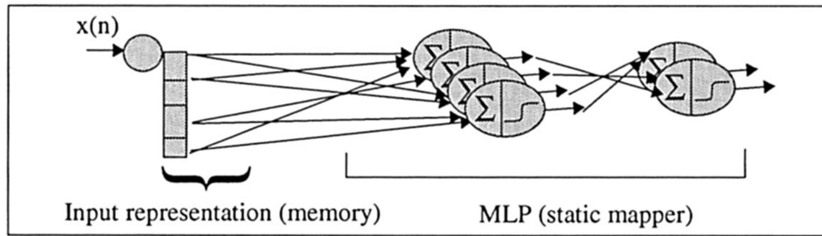

FIGURE 20.9 A focused TLN.

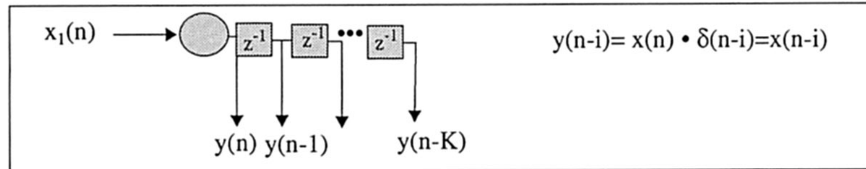

FIGURE 20.10 Tap delay line memory.

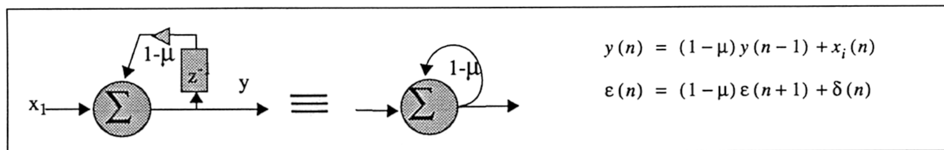

FIGURE 20.11 Memory by feedback (context PE).

It is possible to combine the advantages of memory by feedback with the ones of the memory by delays in linear systems called *dispersive delay lines.* The most studied of these memories is a cascade of low-pass functions called the *gamma memory* [de Vries and Principe, 1992]. The gamma memory has a free parameter μ that controls and decouples memory depth from resolution of the memory. *Memory depth D* is defined as the first moment of the impulse response from the input to the last tap K, while *memory resolution R* is the number of taps per unit time. For the gamma memory $D = K/\mu$, and $R = \mu$; i.e., changing μ modifies the memory depth and resolution inversely. This recursive parameter μ can be adapted with the output MSE as the other network parameters; i.e., the ANN is able to choose the best memory depth to minimize the output error, which is unlike the tap delay memory.

Training-Focused TLN Architectures

The appeal of the focused architecture is that the MLP weights can be still adapted with back-propagation. However, the input/output mapping produced by these networks is static. The input memory layer is bringing in past input information to establish the value of the mapping.

As we know in engineering, the size of the memory is fundamental to identify, for instance, an unknown plant or to perform prediction with a small error. But note now that with the focused TLN the models for system identification become nonlinear (i.e., nonlinear moving average — NMA).

When the tap delay implements the short-term memory, straight back-propagation can be utilized since the only adaptive parameters are the MLP weights. When the gamma memory is utilized (or the context PE), the recursive parameter is adapted in a total adaptive framework (or the parameter is preset by some external consideration). The equations to adapt the context PE and the gamma memory are shown in Figs. 20.11 and 20.12, respectively. For the context PE $\delta(n)$ refers to the total error that is back-propagated from the MLP and that reaches the dual context PE.

© 2000 by CRC Press LLC

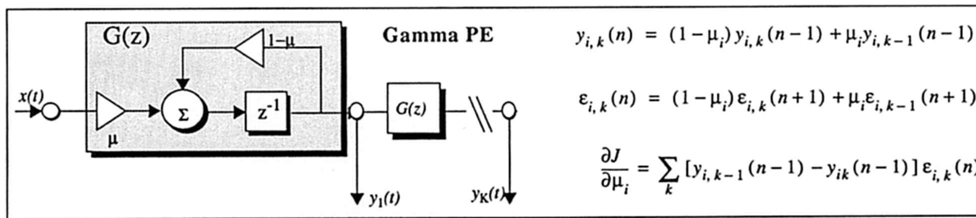

FIGURE 20.12 Gamma memory (dispersive delay line).

20.5 Hebbian Learning and Principal Component Analysis Networks

Hebbian Learning

Hebbian learning is an unsupervised learning rule that *captures similarity* between an input and an output through *correlation.* To adapt a weight w_i using Hebbian learning we adjust the weights according to $\Delta w_i = \eta x_i y$ or in an equation [Haykin, 1994]

$$w_i(n+1) = w_i(n) + \eta\, x_i(n) y(n) \tag{20.12}$$

where η is the step size, x_i is the ith input and y is the PE output.

The output of the single PE is an inner product between the input and the weight vector (formula in Fig. 20.13). It measures the similarity between the two vectors — i.e., if the input is close to the weight vector the output y is large; otherwise it is small. The weights are computed by an outer product of the input X and output Y, i.e., $W = XY^T$, where T means transpose. The problem of Hebbian learning is that it is unstable; i.e., the weights will keep on growing with the number of iterations [Haykin, 1994].

Oja proposed to stabilize the Hebbian rule by normalizing the new weight by its size, which gives the rule [Haykin, 1994]:

$$w_i(n+1) = w_i(n) + \eta\, y(n)\left[x_i(n) - y(n)\, w_i(n)\right] \tag{20.13}$$

The weights now converge to finite values. They still define in the input space the direction where the data cluster has its largest projection, which corresponds to the eigenvector with the largest eigenvalue of the input correlation matrix [Kung, 1993]. The output of the PE provides the largest eigenvalue of the input correlation matrix.

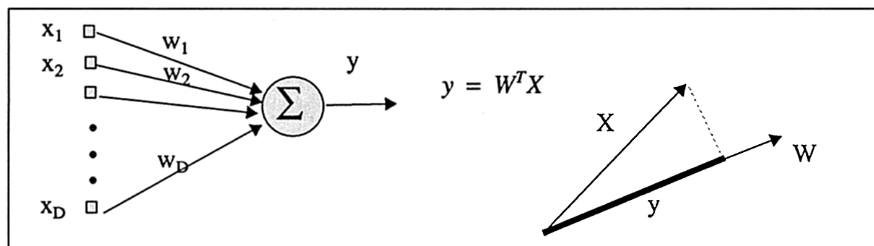

FIGURE 20.13 Hebbian PE.

© 2000 by CRC Press LLC

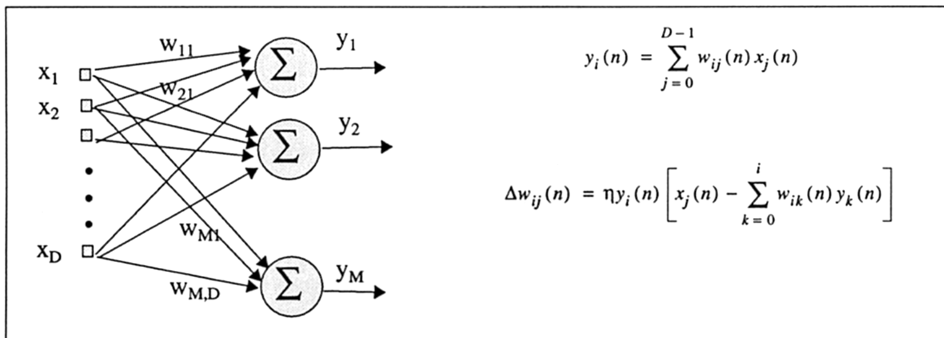

FIGURE 20.14 PCA network.

Principal Component Analysis

Principal component analysis (PCA) is a well-known technique in signal processing that is used to project a signal into a signal-specific basis. The importance of PCA analysis is that it provides *the best linear projection* to a subspace in terms of preserving the signal energy [Haykin, 1994]. Normally, PCA is computed analytically through a singular value decomposition. PCA networks offer an alternative to this computation by providing an iterative implementation that may be preferred for real-time operation in embedded systems.

The PCA network is a one-layer network with linear-processing elements (Fig. 20.14). One can extend Oja's rule for many-output PEs (less or equal to the number of input PEs), according to the formula shown in Fig. 20.14 which is called the Sanger's rule [Haykin, 1994]. The weight matrix rows (that contain the weights connected to the output PEs in descending order) are the eigenvectors of the input correlation matrix. If we set the number of output PEs equal to $M < D$, we will be projecting the input data onto the M largest principal components. Their outputs will be proportional to the M largest eigenvalues. Note that we are performing an eigendecomposition through an iterative procedure.

Associative Memories

Hebbian learning is also the rule to create *associative memories* [Zurada, 1992]. The most-utilized associative memory implements *heteroassociation*, where the system is able to associate an input X to a designated output Y which can be of a different dimension (Fig. 20.15). So, in heteroassociation the signal Y works as the desired response.

We can train such a memory using Hebbian learning or LMS, but the LMS provides a more efficient encoding of information. Associative memories differ from conventional computer memories in several respects. First, they are content addressable, and the information is distributed throughout the network, so they are robust to noise in the input. With nonlinear PEs or recurrent connections (as in the famous Hopfield network) [Haykin, 1994] they display the important property of *pattern completion;* i.e., when the input is distorted or only partially available, the recall can still be perfect.

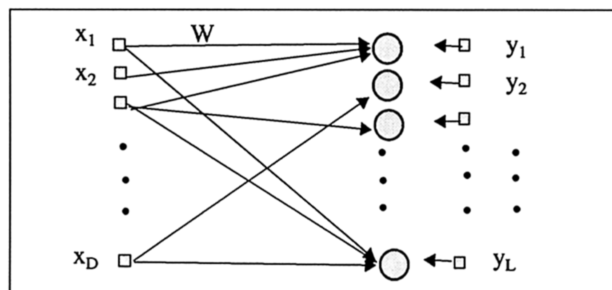

FIGURE 20.15 Associative memory (heteroassociation).

© 2000 by CRC Press LLC

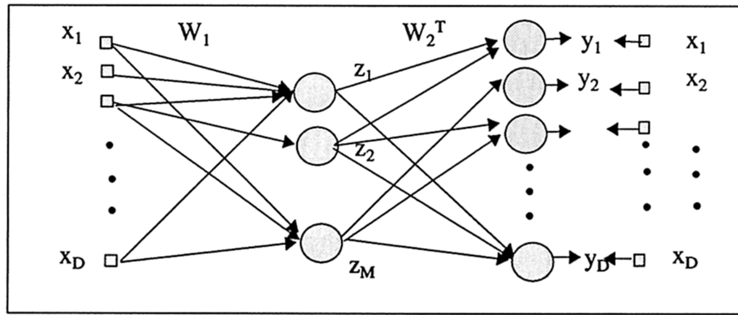

FIGURE 20.16 Autoassociator.

A special case of associative memories is called the *autoassociator* (Fig. 20.16), where the training output of size D is equal to the input signal (also a size D) [Kung, 1993]. Note that the hidden layer has fewer PEs ($M \ll D$) than the input (bottleneck layer). $W_1 = W_2^T$ is enforced. The function of this network is one of *encoding or data reduction*. The training of this network (W_2 matrix) is done with LMS. It can be shown that this network also implements PCA with M components, even when the hidden layer is built from nonlinear PEs.

20.6 Competitive Learning and Kohonen Networks

Competition is a very efficient way to divide the computing resources of a network. Instead of having each output PE more or less sensitive to the full input space, as in the associative memories, in a competitive network each PE specializes into a piece of the input space and represents it [Haykin, 1994]. Competitive networks are linear, single-layer nets (Fig. 20.17). Their functionality is directly related to the competitive learning rule, which belongs to the unsupervised category. First, only the PE that has the largest output gets its weights updated. The weights of the winning PE are updated according to the formula in Fig. 20.17 in such a way that they approach the present input. The step size exactly controls how much is this adjustment (see Fig. 20.17).

Notice that there is an intrinsic nonlinearity in the learning rule: only the PE that has the largest output (the winner) has its weights updated. All the other weights remain unchanged. This is the mechanism that allows the competitive net PEs to specialize.

Competitive networks are used for clustering; i.e., an M output PE net will seek M clusters in the input space. The weights of each PE will correspond to the centers of mass of one of the M clusters of input samples. When a given pattern is shown to the trained net, only one of the outputs will be active and can be used to *label* the sample as belonging to one of the clusters. No more information about the input data is preserved.

Competitive learning is one of the fundamental components of the Kohonen self-organizing feature map (SOFM) network, which is also a single-layer network with linear PEs [Haykin, 1994]. Kohonen learning creates annealed competition in the output space, by adapting not only the winner PE weights but also their spatial

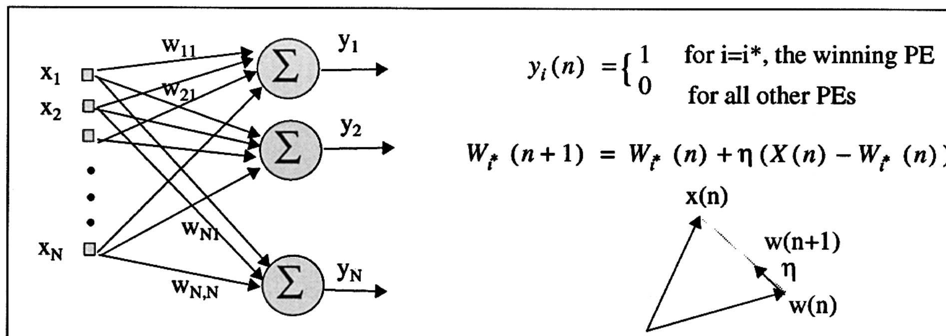

FIGURE 20.17 Competitive neural network.

© 2000 by CRC Press LLC

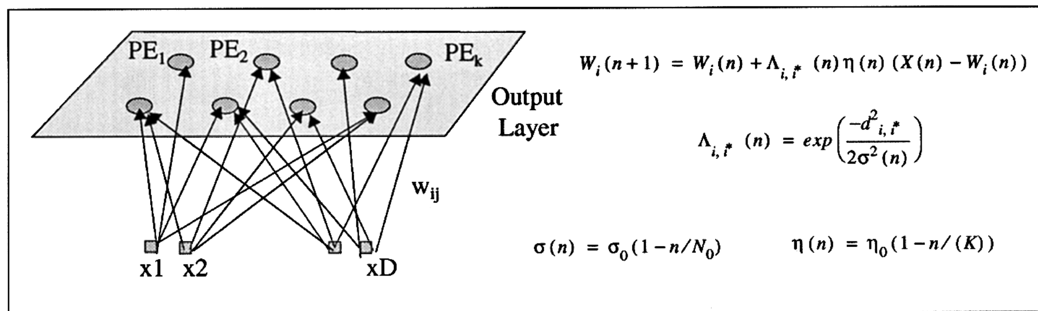

FIGURE 20.18 Kohonen SOFM.

neighbors using a Gaussian neighborhood function Λ. The output PEs are arranged in linear or two-dimensional neighborhoods (Fig. 20.18)

Kohonen SOFM networks produce a mapping between the continuous input space to the discrete output space preserving topological properties of the input space (i.e., local neighbors in the input space are mapped to neighbors in the output space). During training, both the spatial neighborhoods and the learning constant are decreased slowly by starting with a large neighborhood σ_0, and decreasing it (N_0 controls the scheduling). The initial step size η_0 also needs to be scheduled (by K).

The Kohonen SOFM network is useful to project the input to a subspace as an alternative to PCA networks. The topological properties of the output space provide more information about the input than straight clustering.

References

C. M. Bishop, *Neural Networks for Pattern Recognition,* New York: Oxford University Press, 1995.

de Vries and J. C. Principe, "The gamma model — a new neural model for temporal processing," *Neural Networks,* Vol. 5, pp. 565–576, 1992.

S. Haykin, *Neural Networks: A Comprehensive Foundation,* New York: Macmillan, 1994.

S. Y. Kung, *Digital Neural Networks,* Englewood Cliffs, N.J.: Prentice-Hall, 1993.

J. M. Zurada, *Artificial Neural Systems,* West Publishing, 1992.

Further Information

The literature in this field is voluminous. We decided to limit the references to text books for an engineering audience, with different levels of sophistication. Zurada is the most accessible text, Haykin the most comprehensive. Kung provides interesting applications of both PCA networks and nonlinear signal processing and system identification. Bishop concentrates on the design of pattern classifiers.

Interested readers are directed to the following journals for more information: *IEEE Transactions on Signal Processing, IEEE Tranactions on Neural Networks, Neural Networks, Neural Computation,* and *Proceedings of the Neural Information Processing System Conference* (NIPS).

© 2000 by CRC Press LLC