NEURAL NET APPLICATIONS

The most complex computing device in nature recognized at present is the human brain. A computer model that matches the functionality of the brain in a very fundamental manner has led to the development of artificial neural networks (1). These networks have emerged as generalizations of mathematical models of neurobiology based on the assumptions that information processing occurs at many simple elements called neurons; that signals are passed between neurons over connection links; that each connection link has an adaptive weight associated with it that, in a typical neural network, multiplies the signal transmitted; and that each neuron applies an activation function to its net input to determine its output signal.

PRINCIPLES

Inspiration from the Brain

There exists a close analogy between the structure of a biological neuron and that of the artificial neuron or processing element that is the basic building block of an artificial neural

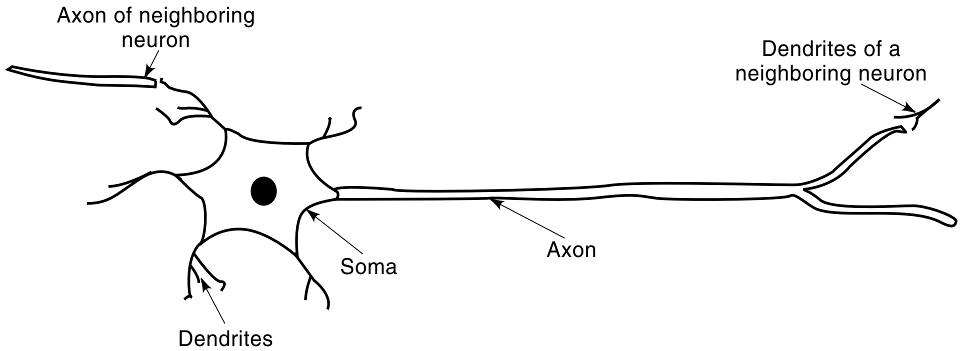

Figure 1. A biological neuron showing the cell body and the axon which transmits action potentials to neighboring neurons via their dendrites.

network (henceforth referred to simply as a neural network). A biological neuron has three types of components that constitute its structure: the dendrites, the soma, and the axon. The many dendrites receive signals from other neurons, which are electrical impulses (action potentials) that are transmitted across a synaptic gap, via the synapses that are located at dendritic ends, by converting electrical energy into chemical energy. The action of the chemical transmitters modifies the incoming signal in a manner similar to the adaptive adjustment of weights in a neural network. The soma (i.e., the cell body) weights and sums the incoming signals. When the sum exceeds a certain threshold, the cell fires; that is, it sends out an action potential over its axon to other cells. The transmission of the action potential signal from a neuron is caused by concentration differences of ions on either side of the neuron's axon sheath. The ions most directly involved are sodium, potassium, and chlorine. A generic biological neuron is illustrated in Fig. 1 together with an axon from a neighboring neuron from which the illustrated neuron can receive input signals and the dendrites of one other neuron to which the illustrated neuron can send signals.

Several key properties of the processing elements of artificial neural networks are suggested by the properties of biological neurons:

1. The processing elements receive many signals as input.
2. Signals are modified by weights at the receiving synapses.
3. The processing elements sum the weighted inputs.
4. For sums above a certain threshold, the neuron transmits a single output.
5. The output from a single neuron may serve as input to many other neighboring neurons.
6. A synapse's strength may be modified by experience.
7. Neurotransmitters may be excitatory or inhibitory.

Another important characteristic that is shared by biological and artificial neural networks is that of fault tolerance. Biological neural networks are primarily fault-tolerant in two respects. First, humans are able to recognize many input signals that are somewhat different from any signal they have seen before. Second, humans have the ability to tolerate damage to the neural system itself. Humans are born with approximately 10^{11} neurons. Most of these neurons are located in the brain, and for the most part are not replaced when they die. In spite of an ongoing loss of neurons, humans continue to learn. Even in cases of traumatic neural loss, other neurons can sometimes be trained to take over the functions of the damaged cells. In a similar manner, artificial neural networks can be designed to be insensitive to minor damage to the physical topology of the network, and can be retrained to compensate for major topological changes or damage.

Model of an Artificial Neuron

In an artificial neural network, the unit analogous to the biological neuron is referred to as the *processing element.* A processing element has several input paths (dendrites), and it combines, usually by a simple summation operation, the values of these input paths. The result is an internal activity level for the processing element. This combined input is modified by a transfer function. The transfer function can be a threshold function that passes information only if the combined activity level reaches a certain value, or it can be a continuous function of the combined input. It is most common to use the sigmoidal family of functions for this purpose. The output value of the transfer function is passed directly to the path leaving the processing element.

The output path of a processing element can be connected to the input paths of other processing elements through connection weights which correspond to the synaptic strengths of the neural connections. Since each connection has a corresponding weight, the signals on the input lines of a processing element are modified by these weights prior to being summed as is illustrated in Fig. 2. Thus, the resulting function is a weighted summation. McCulloch and Pitt (2) proposed a simple model of a neuron as a binary threshold unit. Specifically,

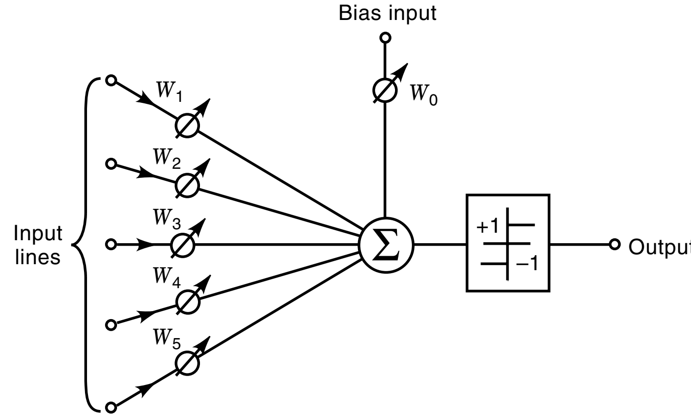

Figure 2. A simple artificial neuron depicting a mathematical model of the biological neuron.

the model of the neuron, as described above, computed a weighted sum of its inputs from other units, and output a one or a zero according to whether this sum was above or below a certain threshold as given by

$$n_i(t+1) = \Theta\left(\sum_j w_{ij} n_j(t) - \mu_i\right) \tag{1}$$

where n_i, which can be either 1 or 0, represents the state of the neuron i as firing or not firing, respectively. The time index t is treated as being discrete, with each processing step being equal to one time step. $\Theta(x)$ is the activation function of the neuron, and in this case is specifically the unit step function given by

$$\Theta(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The weight w_{ij} represents the strength of the synapse connecting neuron j to neuron i and can be either positive or negative, depending on whether it is excitatory or inhibitory. The neuron-specific parameter μ_i is the threshold value for neuron i, and the weighted sum of its inputs must reach or exceed the threshold in order for the neuron to fire.

In itself, this simplified model of a neuron is not very impressive. However, a number of interesting effects result from the manner in which neurons are interconnected.

Typical Network Architectures

An artificial neural network consists of many neurons or processing elements joined together; usually organized into groups called layers. A typical network consists of a sequence of layers with full or random connections between successive layers. There are typically two layers with connections to the outside world—an input buffer where data are presented to the network, and an output buffer that holds the response of the network to a given input pattern. The nodes in the input layer encode the instances or patterns to be presented to the network. The output layer nodes encode solutions to be assigned to the instances under consideration at the input layer. Layers distinct from the input and output buffers are called hidden layers. These layers typically consist of nonlinear units that are used to capture and store the nonlinear representation of the mapping under consideration.

Networks are called *fully connected* if every neuron in a certain layer is connected to every other neuron in a layer in front of it, and *feedforward* if the connections all point in one direction. Network architectures where feedback connections or loops are included are called *recurrent* networks. These are typically used for the processing of temporal information.

An Introduction to Learning

There are two main phases in the operation of a network—learning and recall. The details of these vary from network to network. Learning is the process of changing or modifying the connection weights in response to stimuli being presented at the input buffer and optionally at the output buffer. A stimulus presented at the output buffer corresponds to a desired response to a given input; this response may be provided by a teacher or supervisor. This kind of learning is called *supervised learning* and is by far the most common learning strategy. A network is said to be have been *trained* if it can successfully predict an outcome in response to novel inputs. If the desired output is different from the input, the trained network is called a *heteroassociative* network. If, for all the training examples, the desired output vector equals the input vector, the net is called an *autoassociative* network. Rumelhart, Hinton, and Williams (3) discuss several applications of networks incorporating supervised learning methods.

If no desired output is shown, the learning is called *unsupervised* learning. Learning occurs by *clustering,* i.e., the detection of structure in the incoming data. Kuperstein (4) has implemented a neural controller in a five-degree-of-freedom robot to grasp objects in arbitrary positions in a three dimensional world. This controller, called INFANT, learns visual–motor coordination without any knowledge of the geometry of the mechanical system and without a teacher. INFANT adapts to unforeseen changes in the geometry of the physical motor system, to the internal dynamics of the control circuits, and to the location, orientation, shape, weight, and size of objects.

A third kind of learning falls in between the above two modes. This is called *reinforcement* learning. Here a *critic* appropriately rewards or penalizes the learning system until it ultimately produces the correct output in response to a given input pattern. Anderson (5) has simulated an inverted pendulum as a control task with the goal of learning to balance the pendulum without a priori knowledge of the dynamics. Performance feedback is assumed to be available only as a failure signal when the pendulum falls or reaches the bounds of a horizontal track. Whatever the kind of learning used, an essential characteristic of any adaptive network is its learning rule, which specifies how weights change in response to a learning example. Learning may require iterating the training examples through the network thousands of times. The parameters governing a learning rule may change over time as the network progresses in its learning.

Another important phase in the operation of a network is termed *recall.* Recall refers to the manner in which the network processes a stimulus presented at its input buffer and creates a response at the output buffer. Often, recall is an integral part of the learning process, as when the desired response of a network must be compared with the actual output of the network to create an error signal. The recall phase is used to gauge whether a network has learned to perform the specified task or not. A network is said to have learnt to perform a specified task if a predefined (task specific) objective function has been minimized successfully.

The Perceptron

A single-layered perceptron consists of an input and an output layer. It is a direct extension of the biological neuron described previously. The activation function, as shown in Fig. 2, is a hard-limiting signum function. The output unit will assume the value +1 if the sum of its weighted inputs is greater than its threshold. Hence an input pattern will be classified into category A at the output unit j using

$$\sum W_{ij} X_i > \Theta_j \tag{3}$$

where W_{ij} is the weight from unit i to unit j, X_i is the input from unit i, and Θ_j is the threshold on unit j (note that in Fig. 2, the subscript j is omitted, since only one output unit is taken into consideration). Otherwise, the input pattern will be classified into category B. The perceptron learning algorithm can be described as follows:

Initialize all the weights and thresholds to small random numbers. The thresholds are negatives of the weights from the bias unit, whose input level is fixed at +1. The activation level of an input unit is determined by the pattern presented to it. The activation level of an output unit is given by

$$O_j = F_{\mathrm{h}}\left(\sum W_{ij}X_i - \Theta_j\right) \tag{4}$$

where W_{ij} is the weight from an input X_i, Θ_j is the threshold, and F_{h} is the hard-limiting activation function:

$$F_{\mathrm{h}}(p) = \begin{cases} +1, & p > 0 \\ -1, & p \leq 0 \end{cases} \tag{5}$$

The weights are adjusted by

$$W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij} \tag{6}$$

where $W_{ij}(t)$ is the weight from unit i to unit j at time t and ΔW_{ij} is the weight adjustment for iteration step at time $t + 1$. The weight change may be computed by the delta rule:

$$\Delta W_{ij} = \eta \delta_j X_i \tag{7}$$

where η is the learning rate and takes on values between 0 and 1, and δ_j is the error at unit j given by

$$\delta_j = T_j - O_j \tag{8}$$

where T_j is the target output activation and O_j is the actual output activation at the output unit j. The above steps are iterated until convergence is achieved, i.e. the actual output activation (classification) is the same as the target output activation. According to the perceptron convergence theorem (6), if the input data points are linearly separable, the perceptron learning rule will converge to a solution in a finite number of steps for any initial choice of the weights.

Linear Versus Nonlinear Separability

Consider a case where a perceptron has n inputs and one output. Hence the perceptron equation

$$\sum_{i=1}^{n} W_{ij}X_i = \Theta_j \tag{9}$$

forms a hyperplane in the $(n + 1)$-dimensional input space, dividing the space into two halves. When $n = 2$, the hyperplane is reduced to a line. Linear separability refers to the case where a linear hyperplane exists that can separate the patterns into two distinct classes. Unfortunately, most classification problems fall in the category of problems requiring a nonlinear hyperplane to separate the patterns into their distinct classes. A good example is the XOR logic problem, which is nonlinearly separable, whereas its counterpart, the AND, is linearly separable. While the XOR solution requires a nonlinear curve to separate its zero output class from its one output class, the AND can be solved using a straight line. This is illustrated in Fig. 3. In essence, a multilayered perceptron (modern day neural network) is required to solve classification problems that are not linearly separable.

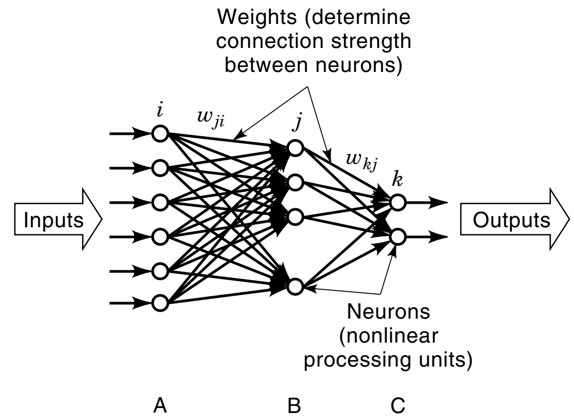

Figure 3. Linear (AND) versus nonlinear (XOR) separability.

Multilayered Perceptrons and the Backpropagation Algorithm

A typical feedforward network topology showing a multilayered perceptron is illustrated in Fig. 4. A multilayered perceptron is a feedforward neural network with at least one hidden layer. It can deal with nonlinear classification problems, since it can form complex decision regions—unlike simple perceptrons, which were restricted to hyperplanes. The figure shows a three-layered network with one hidden layer, but in principle there could be more than one hidden layer to store the internal representations. The fundamental concept underlying the design of the network is that the information entering the input layer is mapped as a nonlinear internal representation in the units of the hidden layer(s), and the outputs are generated by this internal representation rather than by the input vector. Given enough hidden units, input vectors can be encoded in a format that ensures generation of the desired output vectors.

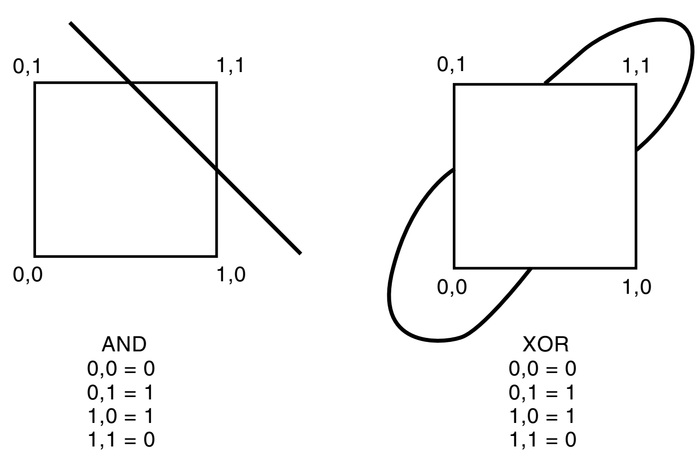

Figure 4. A typical multilayered feed-forward network topology where output = w_{kj} * sigmoid (w_{ji} * input).

As is evident from Fig. 4 the outputs of the units in layer A are multiplied by appropriate weights w_{ji}, and these are fed as inputs to the hidden layer. Hence, if o_i are the outputs of units in layer A, then the total input to the hidden layer (layer B) is

$$\mathrm{net}_B = \sum_i w_{ji} o_i \tag{10}$$

and the output o_j of a unit in layer B is

$$o_j = f(\mathrm{net}_B) \tag{11}$$

where f is the nonlinear activation or transfer function. It is a common practice to choose the sigmoid function given by

$$f(x) = \frac{1}{1 + e^{-x}} \tag{12}$$

as the nonlinear activation function. However, any input–output function that possesses a bounded derivative can be used in place of the sigmoid function (3).

The aim when using a neural network is to find a set of weights that ensures that for each input vector the output vector produced by the network is the same as (or sufficiently close to) the desired output vector. If there is a fixed, finite set of input–output pairs, the total error in the performance of the network with a particular set of weights can be computed by comparing the actual and the desired output vectors for each presentation of an input vector.

The error at any output unit e_k in layer C is

$$e_k = t_k - o_k \tag{13}$$

where t_k is the desired output for the unit in layer C, and o_k is the actual output produced by the network. A measure of the total error E at the output may be defined as

$$E = \frac{1}{2} \sum_k (t_k - o_k)^2 \tag{14}$$

Learning is accomplished by changing network weights so as to minimize the error function. To minimize E by gradient descent, it is necessary to compute the partial derivative of E with respect to each weight in the network. This is the sum of the partial derivatives for each of the input–output pairs (7). The forward pass through the network, where the units in each layer have their states determined by the inputs they receive from the units in the previous layers, is quite straightforward. The backward pass through the network, which involves the backpropagation of weight error derivatives (i.e., the supervisory learning information) from the output layer back to the input layer, is more complicated.

For the sigmoid activation function, the so-called delta rule (8) for iterative convergence toward a solution may be stated in general as

$$\Delta w_{kj} = \eta \delta_k o_j \tag{15}$$

where the parameter η is called the learning rate parameter (3). The error δ_k at an output layer unit k is given by

$$\delta_k = (t_k - o_k) o_k (1 - o_k) \tag{16}$$

and the error δ_j at a hidden-layer unit is given by

$$\delta_j = o_j (1 - o_j) \sum_k \delta_k w_{kj} \tag{17}$$

In practice, it has been found that one strategy to speed up the convengence without causing oscillations is to modify the delta rule for the sigmoid function as given above by including a momentum term given by

$$\Delta w_{kj}[p + 1] = \eta \delta_k o_j + \alpha \, \Delta w_{kj}[p] \tag{18}$$

where the index p indicates the presentation iteration number, or the number of times a set of input vectors has been presented to the network. The momentum factor α is an exponential decay factor having a value between 0 and 1 that determines the relative contribution of the current gradient and the earlier gradients to the weight change.

Practical Issues Relating to the Backpropagation Algorithm

In the past few years, the backpropagation algorithm has proven to be the most popular of all learning algorithms, as evidenced by commercial as well as academic use (9,10). Given this enormous interest, a lot of effort has been devoted to determining improvements and modifications to the original version of the algorithm together with the identification of key issues to pay attention to when using the algorithm. Given below is an overview of some of the prominent issues and modifications. The interested reader can learn further by studying the references cited in this section.

The basic backpropagation algorithm is quite slow, and many variations have been suggested to make it faster. Other goals have been to improve the generalizational ability and the avoidance of local minimum traps in the error surface. Some authors have employed alternative cost functions as opposed to the quadratic cost function used in the original version. Others have considered transforming the data using transforms such as wavelets, fast Fourier transforms, and simple trigonometric, linear, and logarithmic transformations. Also, as mentioned previously, the addition of the momentum parameter enhances the speed considerably, especially in situations where one has cost surface valleys with steep sides but a shallow slope along the valley floor. The idea is to give each connection weight some inertia or momentum such that it tends to change in the direction of the average downhill “force” that it feels, instead of oscillating wildly with every little kick in the learning rate parameter.

To choose appropriate learning rate and momentum parameter values for a given problem is not a straightforward matter. Moreover, the best values at the beginning of the training may not be so good later on in the process. Hence many authors have suggested adjusting these parameters automatically, as the learning progresses (11). One could even have separate parameter sets for each connection and modify them according to whether a particular weight update did actually decrease the cost function (12).

Although gradient descent is one of the simplest optimization techniques, it is not necessarily the best approach for all problems. Instead of considering the slope of the error surface (first derivative or Jacobian information), many authors have worked with its curvature (second derivative or Hessian information). While this offers higher accuracy, there is a tradeoff with regard to computational effort, given that one needs to invert an N by N Hessian matrix at every iteration, taking on the order of N^3 steps every time. Hence this method is optimal for use with small problems. Other authors (10) have considered other ways to approximate the Hessian algebraically or ways to avoid the need to invert it at every step.

The very best practical algorithms still employ first-derivative information, but strengthen that with efficient line search procedures that move along the error surface with adaptive step sizing and directional vectors. The conjugate gradient methods fall under this category and are among the most practical methods for solving real world problems. Hence, given the task of error minimization, deciding on which direction to move in at each step and determining how much to move at each step are the two basic issues to be considered when developing variant algorithms. Following are some of the other issues that relate to the backpropagation algorithm and that are critical to obtaining improved network performance:

Generalization. This is concerned with how well the network performs on the problem with respect to both seen and unseen data. It is usually tested on new data outside the training set. Generalization is dependent on the network architecture and size, the learning algorithm, the complexity of the underlying problem, and the quality and quantity of the training data. Research has been conducted to determine factors such as the number of training patterns required for good generalization and the optimal network size, architecture, and learning algorithm. Vapnik and Chervonenkis (13) showed that it was possible to compute a quantity called VCdim of a network that enabled the computation of the number of training patterns required for a good generalization for that network.

Network Pruning. In a fully connected network, generally there is a large amount of redundant information encoded in the weights. This is because of the heuristic, nonparametric manner in which the choice of the number of hidden units is made when setting up the network topology for the solution of a particular problem. Thus, it is possible to remove some weights without affecting network performance, and this reduction improves the generalizational properties and lowers the computational burden of the network. It also ensures a solution that employs a topology with degrees of freedom consistent with that of the natural system being approximated. Such methods evaluate the *saliency* of every hidden unit and perform a rank ordering to make a decision on weight elimination (14). Another method to help prune networks is to give each weight the tendency to decay to zero unless reinforced and strengthened by incoming patterns.

Network Construction Algorithms. Rather than starting with too large a network and then pruning, work has been reported in the literature (15) where researchers have started with small networks and then used the training data to gradually grow the network to an optimal size.

Local Minima. Gradient descent and all other optimization techniques can become stuck at local minima of the cost function. Although local minima have not been too much of a problem in most cases studied empirically, still one needs to be aware of their existence and develop a capability for detecting and allowing for their presence. The magnitudes of the initial weights are very important in this regard. Perturbation techniques (14) such as annealing and random dithering have been studied by researchers as being effective countermeasures for local minima.

The backpropagation algorithm falls in a class of learning algorithms termed *globally* generalizing or approximating. The fundamental problem with global approximation paradigms is that they are susceptible to *global network collapse* when attempting to perform on-line learning. It is caused by a lack of *persistence of excitation* to cause the control parameters within the learning paradigm to be updated after the system settles into a desired state. Local approximation strategies (16) on the other hand, simply learn "pockets of the model" and do not generalize over the entire model, which prevents global network collapse. The other major problem when working with on-line learning tasks is the ubiquitous presence of noise in the incoming sensor data. Global learning paradigms change all the weights in the network in response to incoming data. Hence, highly noisy signals can cause complete global degradation of the model represented within the network. Local learning paradigms work around this problem, since noisy data will cause damage only in portions of the network and not degrade the entire learned representation.

Another advantage of local paradigms is that they are computationally efficient, since only a small portion of the weight space is updated in response to a control input at any given time. They also display rapid convergence, as learning is local and is performed in distinct "pockets" of the systems dynamics. They do not encounter the problem of local minima, since the local error surface is quadratic. They do require large amounts of memory (17,18), but then again, lack of sufficient memory is scarcely an issue in the current era in which inexpensive, fast, short-access-time memory modules are commonplace in computer systems.

The cerebellar model articulation controller (CMAC) (19) and radial basis function (RBF) networks (20) belong to the class of locally generalizing algorithms. An RBF network is a one-hidden-layer network whose output units form a linear combination of the basis functions computed by the hidden units. The basis functions in the hidden layer produce a localized response to the input and hence operate within a localized receptive field. The most commonly used basis function is the Gaussian function, where the output of a hidden unit j is given by

$$O_j = \exp\left(-\frac{(X - W_j)\cdot(X - W_j)}{2\sigma_j^2}\right) \tag{19}$$

where X is the input vector, W_j is the weight vector associated with hidden unit j, and σ^2 is the normalization factor of the Gaussian basis function.

Learning Temporal Sequences

The backpropagation algorithm as described in the previous sections has established itself as the most popular learning rule in the design of neural networks. However, a major limitation of the standard backpropagation algorithm is that it can only learn an input–output mapping that is static. Static mapping is well suited for pattern recognition applications where both the inputs and the outputs represent spatial patterns that are independent of time. But how does one extend the design of a multilayered perceptron so that it assumes a time-varying form and therefore will be able to deal with time-varying signals? For a network to be able to capture dynamic maps, it must be given memory (21). One way to do this is to introduce time delays into the topology of the network and adjust their values during the learning phase. A *time delay neural network* is a multilayered feedforward network whose hidden and output neurons are replicated across time as recurrent connections. The popular training approach is the backpropagation-through-time algorithm (22), which may be derived by unfolding the temporal operation of the network into a standard multilayered feedforward network, the topology of which grows by one layer at each time step.

A Food Dryer Example

Controlling a complex industrial process can be a challenging and appropriate task for a neural network, since rules are often difficult to define, historical data are plentiful but noisy, and perfect numerical accuracy is not required (23). Neural networks can be shown to be extremely efficient in solving mathematically hard classification, prediction, and process control problems (24,25).

Neural networks have been used quite extensively during the last half decade for generating solutions to real world problems. Some application areas are financial forecasting and portfolio management; credit card fraud detection; character and cursive handwriting recognition (26); quality control in manufacturing; process control in industries such as semiconductors, petrochemical, metals, and food; robotics (27,28); medical applications such as ECG, EEG, MRI, and x-ray data classification; drug structure prediction; and in biological systems modeling and modeling applications (29) such as the study of low back pain.

An example is shown in Fig. 5, where a rotary dryer is depicted. Dryers are among the most ubiquitous pieces of industrial equipment. They are commonly employed in the food industry to dry various materials, from corn to onions and garlic. The objective is to dry the food so that its moisture content lies within a certain prespecified band. Hence, the dryer controller uses continuous feedback from moisture meters to control the various input parameters such as feed rates and burner temperatures. However, moisture sensors are extremely unreliable and highly susceptible to clogging and drift. In this situation a *virtual sensor* based on a temporal, dynamic neural network model of the dryer can be a reliable alternative to effect control. The virtual sensor is based on historical data collected as a result of a good set of experiments that dictate the different variables to be collected, their ranges, and the sampling frequencies. Once the sensor model is built, it is validated with novel data that it has never seen before. After validation, the sensor is integrated with the online control loop, typically into the ladder logic of multiple PID (proportional plus integral plus derivative) control loops on standard industrial programmable logic controllers (PLC).

Use of Critics in Reinforcement Learning

The successful control of dynamic systems typically requires considerable knowledge of the systems being controlled, including an accurate model of the dynamics of the system and an accurate expression of the system's desired behavior, usu-

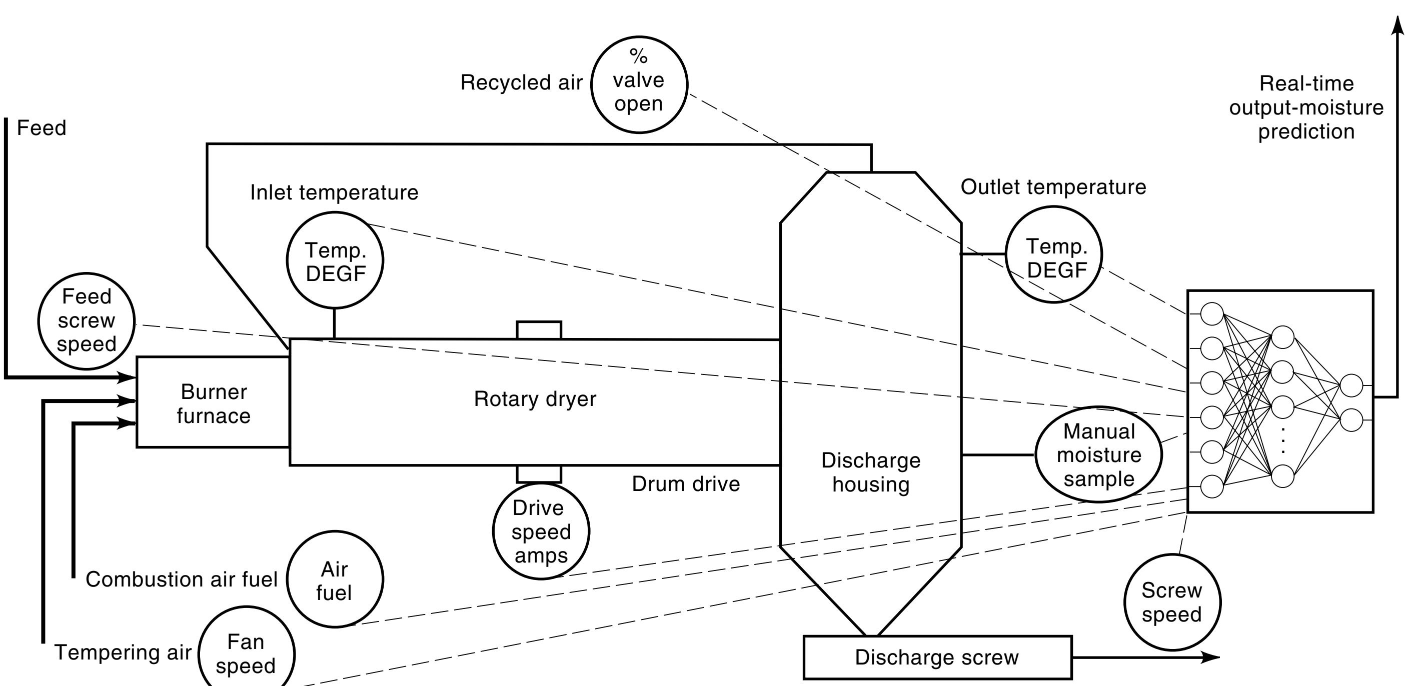

Figure 5. Using a neural network as a virtual sensor in a rotary dryer for real-time output-moisture prediction.

ally in the form of an objective function (30). In situations where such knowledge does not exist, reinforcement learning techniques (31) can be used. Each application of a control action by the reinforcement learning controller results in a qualitative feedback from the environment, which indicates the consequences of that action (and possibly the previous actions), but the feedback does not contain any gradient information to indicate what control actions should be used so that the feedback improves (as in supervised learning). Thus, reinforcement learning can be described as a problem of *credit assignment,* that is, based on the sensor–action–feedback sequences, how to determine what part of the learning system's reasoning process is to be credited (punished or rewarded) and how. This is done by means of a critic or evaluator. The extent to which a local decision or action is credited depends on how it correlates with the reinforcement (i.e., feedback) signal. If enough samples are taken, the noise caused by the variations in other variables is averaged out, and the effect due to a single variable becomes evident. Therefore, with a sufficiently long learning process, an optimal probability can be learned for every local variable.

Reinforcement learning tasks commonly occur in optimal control of dynamic systems and planning problems in artificial intelligence, and the techniques used are closely related to conventional techniques from optimal control theory, as was established by the pioneering work of Barto, Sutton, and Watkins (32). In contrast to backpropagation (or supervised) learning, reinforcement learning does not involve the computation of derivatives and hence lacks gradient information. This feature makes it suitable for application to complex problems where derivative information is hard to obtain. On the other hand, reinforcement learning is very inefficient in large systems. In addition, the system optimization parameters can get locked at a local optimum.

An Inverted Pendulum Example

The inverted pendulum is a classic example of an inherently unstable system. Its dynamics forms the basis for many applications such as gait analysis and control of rocket thrusters. The inverted pendulum task involves the control of a pendulum hinged to the top of a wheeled cart that travels along a track as shown in Fig. 6. The motion of the cart and of the pendulum are constrained to a vertical plane. The state of the system at time t is specified by four real-valued variables: the angle between the pendulum and the vertical, θ; the corresponding angular velocity $\dot{\theta}$; the horizontal position x along the track; and its corresponding velocity $\dot{x}$.

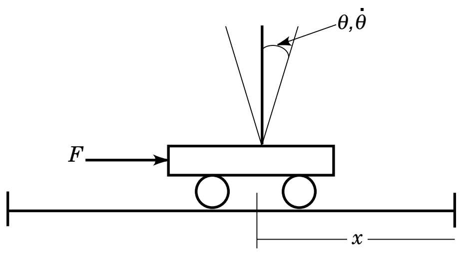

Figure 6. An inverted pendulum system that is controllable using a reinforcement learning scheme.

The goal of this task is to apply a sequence of forces F, of fixed magnitude but variable direction, to the cart such that the pendulum is balanced and the cart does not hit the edge of the track. Zero-magnitude forces are not permissible. Note that reinforcement learning methods are applied to this problem under the assumption that the system dynamics are unknown and that an analytical form of the objective function is unavailable. Bounds on θ and x specify the states of the system for which failure signals can be provided. Two networks, an action network and an evaluation network (5), function together to solve this problem using a temporal-difference reinforcement learning scheme.

Unsupervised Learning Methods

In unsupervised learning, no teacher or supervision exists. Networks falling under this category still have inputs and outputs, but there is no feedback from the environment to indicate what the outputs should be (as in supervised learning) or whether they are correct (as in reinforcement learning). The network must discover for itself patterns, features, correlations, or categories in the input data stream and report the findings as the outputs. Hence, such networks possess the quality of self-organization. Unsupervised learning methods are applicable in problem domains where data are plentiful and redundant, with very little a priori process knowledge, and for dealing with unexpected and changing situations that lack mathematical descriptions. Classification of astronomical data from radio telescopes can be a good unsupervised learning problem.

There are two classes of unsupervised learning algorithms. In the first class, which is based on Hebbian learning, multiple output units are often active together in collective response to the patterns presented at the inputs. In the second class of algorithms, which is based on competitive learning, only one output unit in the entire network, or one unit per prespecified group of output units, fires in response to a pattern presented at the inputs. The output units compete for being the one to fire, and are therefore also referred to as winner-take-all units. One popular competitive learning algorithm is Kohonen's self-organizing feature map (20), which has found application for data compression and vector quantization in two- and three-dimensional signal processing.

A Fault Diagnosis Example

Unsupervised learning methods also find applicability in the area of fault detection and diagnosis. Depicted in Fig. 7 is the schematic block diagram of Neural Applications Corporation's neural network based prototypical system for Catastrophy Management in uptime-critical computer networks at small and medium sized business organizations. Unsupervised learning methodologies are used for performing tasks such as failure mode detection and predictive maintenance. The general idea is to utilize historical data to recognize and cluster trends and to isolate them as faults or failure modes.

NEURAL NETWORK APPLICATIONS IN THE REAL WORLD

Applications in Business, Science, and Industry

Only a few years ago, the most widely reported neural network application outside the financial industry was the air-

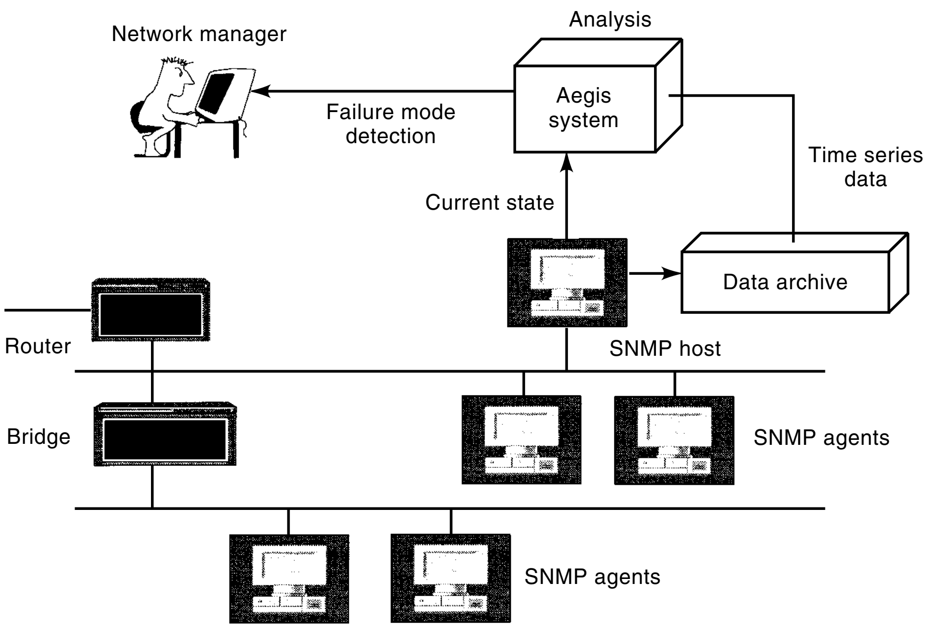

Figure 7. Using neural networks for predictive maintenance and fault diagnostics in computer networks.

port baggage explosive detection systems developed by Science Applications International Corporation (33). Since that time, a large number of industrial and commercial applications have been developed, but the details of most have been shrouded as corporate trade secrets. This growth in the number of applications is due to an increase in the accessibility of computational power and the enhanced availability of commercial software packages that can be quickly tailored to provide low-cost turnkey solutions to a broad spectrum of applications (33). Given below are case studies of two such applications to provide a sample of the variety of possible applications using this technology.

Case Study of the Green Sand Problem in an Automotive Foundry

Molding technology is employed in automotive foundries to cast critical parts such as engine blocks. Typically, a high-pressure green sand molding unit is supplied with *prepared* molding sand by two continuous mullers. The characteristics of the preparation are determined by measures of compaction, green strength, and discharge sand moisture. These measurements are made both by a procedural test in a laboratory every few hours and by an automatic testing unit (if available) every couple of minutes. Based on these measurements, one computes *process measures:* compaction, the available bond in the sand, and the water-to-clay ratio in the sand. The optimization problem, then, is to determine every few seconds the correct rate of water addition (typically in liters per minute) and bond addition (typically in kilograms per minute) such that the measured process measures are as close to the desired process measures as possible. The conventional control method can be termed *reactive control,* i.e., pure feedback control.

The existing control scheme is improved by using a control scheme that can be termed *predictive control,* which is a combination of feedback and feedforward control. On-line process data are used to build a real-time muller model that adapts to muller dynamic state changes. This model is used to make predictions (in times on the order of seconds, rather than minutes as previously done by the automatic tester) and evaluate suggested control responses. This lookahead scheme enables faster control responses to bond needs to support an agile mold production schedule.

The green sand process optimizer is implemented on an Intel Pentium-133 personal computer. It uses an Allen-Bradley 1784-KT communications card and Rockwell RSLINX communication software to transmit data back and forth to an Allen-Bradley PLC-5 via the Data Highway Plus network. The optimizer programs are implemented in a combination of Visual Basic and Visual C++ using Neural Applications Corporation's AEGIS® intelligent systems toolkit. The man–machine interface provides a medium for communication between the program and the process engineers, informing them of key operational data. Process data are collected by the green sand process optimizer system to build the process model and also to implement the alarm generation scheme. Data filtering and statistical analysis are performed to separate out the important variables from the irrelevant ones and to further group the relevant variables as control variables, process model input state variables, and process model output state variables. The model can be described as a fully connected, multilayered time-series neural network. This model is used in the *on-line control mode* to provide dynamic state predictions 90 seconds into the future, which are used to compute the process measures that are sent to the controller. A second use is in the *off-line what-if mode.* This mode is used to perform variable-sensitivity analysis to learn process input–output characteristics using *test profiles.* It allows the system to serve as a low-cost, high-accuracy process simulator. The controller performs constrained optimization using the predictive muller model. It computes optimal values of the water addition (in liters per minute) and bond addition (in kilograms per minute), roughly every 10 seconds, such that the error between measured and predicted process measures of compaction, bond availability in the sand, and water-to-clay ratio in the sand is minimized while adhering to system alarm constraints and boundary conditions. A screen from the program (Fig. 8) shows a live pictorial representation of the system.

Results from a completed installation at the John Deere Foundry in Waterloo, IA indicate a 32% overall decrease in process variability. A set of "soft benefits" such as better loop closure for operations management, real-time visualization and distributed access, and the implementation of a modular PC-based optimization system was also achieved.

Case Study of an Optimization System for an Electric Arc Furnace in a Steel Mill

In the United States steel industry, the total annual electrical energy consumption by electric arc furnaces (EAFs) is 16×10^9 kW · h, at a cost of $600 million. Currently, the primary source of themal energy in EAFs is the electric arc (65%), with other energy input from oxy–fuel burners (5%), and other exothermic reactions (30%) that are supported by injecting oxygen into the furnace. Typically, energy input profiles are developed through trial and error, simple linear algorithms, or the experience of furnace operators.

A neural-network-based optimization system has been developed by Neural Applications Corporation (34) that continually learns to adapt its control of the furnace to correct for changes in scrap makeup, electrode size, system supply voltage, etc. It constantly reoptimizes the control criteria and provides the following two major features. First, it is "three-phase aware" in that it takes into account the effect that an electrode positioning signal will have on the correlation among all the three system phases. The three output signals are chosen so that all three phases meet desired operating conditions. This drastically reduces the setpoint hunting observed in traditional controllers. Second, it continually predicts event occurrences 100 to 300 ms ahead of time, and then sends electrode positioning signals to correct in advance the errors that are anticipated. This causes unprecedented smoothness in operation.

A production version of the system has been installed at 33 different customer locations all over the world, and the consumption of electric power has been reduced by 5 to 8% (an average furnace has a capacity of 30 MW or more, enough power for a city of 30,000 people), wear and tear on the furnace and electrodes has been reduced by 20%, and the daily throughput of steel has been increased, often by 10% or more. The final observation is that this neural-network-based controller increases productivity and yields tremendous cost savings by decreasing electrode consumption, power-on time, and the amount of energy used per ton of steel produced. The natural extension (work in progress) to this success is to investigate the use of similar intelligent technologies for optimization and coordination of all three major energy sources.

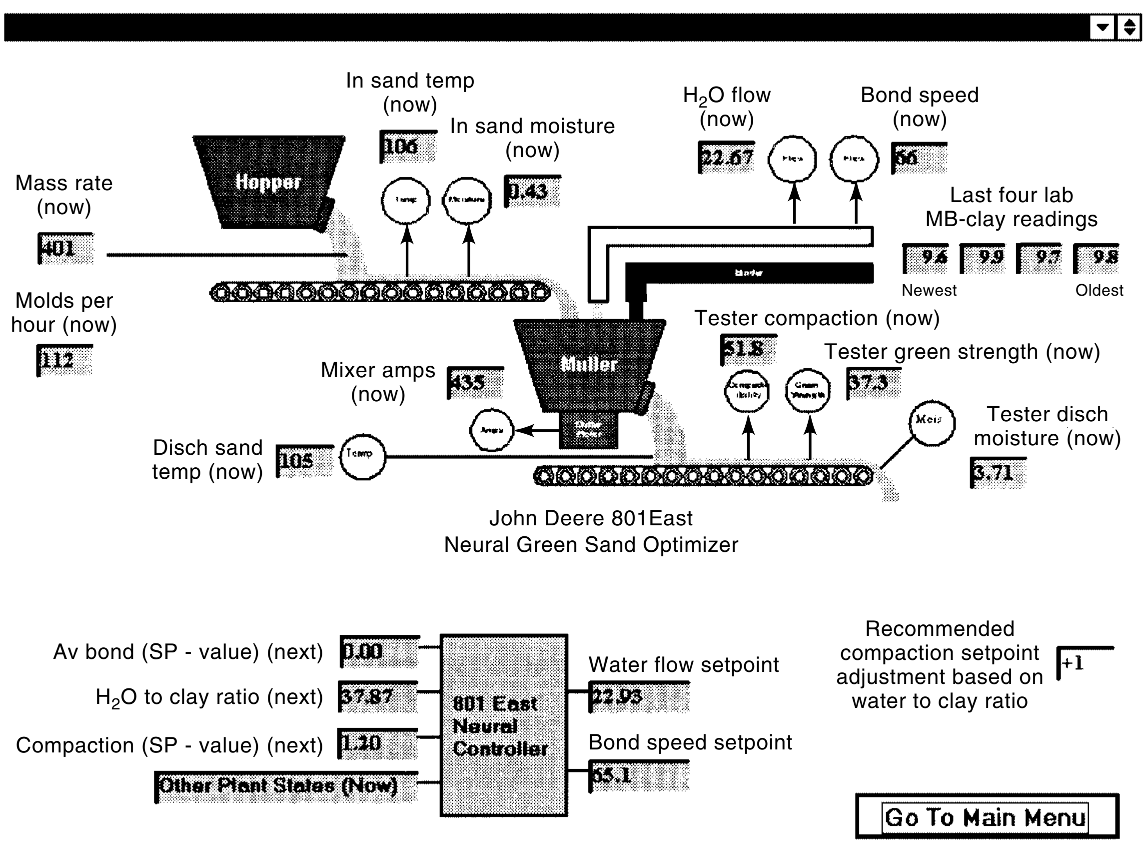

Figure 8. Screen shot from a PC-based system for green sand optimization at an automotive foundry.

SUMMARY

Neural networks and other intelligent techniques such as fuzzy logic and genetic algorithms will probably never be able to compete with conventional techniques at performing precise numerical operations in solving well-defined problems. However, there are a large class of problems in the real world that often involve ambiguity and uncertainty, high noise levels, strong nonlinearity, large numbers of inputs and outputs, etc., and that seem to be more amenable to solutions by neural networks than by conventional means, which usually involve compromising linear approaches as solutions to nonlinear problems.

Intelligent techniques should be used with care, and one should always keep the problem-pull–technology-push tradeoff in mind. They ideally serve to augment an engineer's toolbox so that solutions can be constructed by mixing and matching the strongest technologies as applicable to particular problems. However, based on the current extent of the field, and the rapidity of its growth, it seems reasonable to expect that before the turn of the century, neural networks, fuzzy logic, and genetic algorithms will become household words and a part of day-to-day life.

BIBLIOGRAPHY

1. P. K. Simpson, *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations,* Oxford, UK: Pergamon, 1990.
2. W. S. McCulloch and W. A. Pitt, Logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys.,* **5**: 115–133, 1943.
3. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by backpropagating errors, *Nature,* **323**: 533–536, 1986.
4. M. Kuperstein, INFANT neural controller for adaptive sensory-motor coordination, *Neural Netw.,* **4**: 131–145, 1991.
5. C. W. Anderson, Learning to control an inverted pendulum using neural networks, *IEEE Control Syst. Mag.,* **9**: 31–37, 1989.
6. M. L. Minsky and S. A. Papert, *Perceptrons,* Cambridge, MA: MIT Press, 1969.
7. P. J. Werbos, Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. Dissertation, Harvard University, 1974.
8. B. Widrow and S. D. Stearns, *Adaptive Signal Processing,* Englewood Cliffs, NJ: Prentice Hall, 1985.
9. M. S. Ali and S. Ananthraman, The emergence of neural networks, *Chem. Process.,* September 1995, pp. 30–34.
10. L. Fu, *Neural Networks in Computer Intelligence,* New York: McGraw-Hill, 1994.
11. A. A. Minai and R. J. Williams, Backpropagation heuristics: A study of the extended delta-bar-delta algorithm, *Int. Joint Conf. Neural Netw.,* **1**: 595–600, 1990.
12. S. Huang and Y. Huang, Learning algorithms for perceptrons using back-propagation with selective updates, *IEEE Control Syst. Mag.,* April, 1990, pp. 56–61.
13. V. N. Vapnik and A. Y. Chervonenkis, On the uniform convergence of relative frequencies of events to their probabilities, *Theor. Probab. Appl.,* **17**: 264–280, 1971.
14. J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation,* Reading, MA: Addison-Wesley, 1991.
15. S. E. Fahlman and C. Lebiere, The cascade-correlation learning architecture, in *Advances in Neural Information Processing Systems 2,* Morgan Kaufmann, 1990, pp. 524–532.
16. S. Ananthraman and D. P. Garg, Training backpropagation and CMAC neural networks for control of a SCARA robot, *Eng. Appl. Artif. Intell.,* **6** (2): 105–115, 1993.
17. P. Kanerva, *Sparse Distributed Memory,* Cambridge, MA: MIT Press, 1988, pp. 5–60.
18. L. G. Kraft and D. P. Campagna, A comparison between CMAC neural network control and two traditional adaptive control systems, *IEEE Control Syst. Mag.,* 1990, pp. 36–43.
19. J. S. Albus, A new approach to manipulator control: The cerebellar model articulation controller (CMAC), *Trans. ASME J. Dynamic Syst. Meas. Control,* **97** (3): 220–227, 1975.
20. S. Haykin, *Neural Networks: A Comprehensive Foundation,* London: Macmillan College Publishing, 1994.
21. K. J. Hunt et al., Neural networks for control systems—a survey, *Automatica,* **28** (6): 1083–1112, 1992.
22. D. H. Nguyen and B. Widrow, Neural networks for self-learning control systems, *Int. J. Control,* **54** (6): 1439–1451, 1991.
23. K. S. Narendra and K. Parthasarathy, Identification and control of dynamical systems using neural networks, *IEEE Trans. Neural Netw.,* **1**: 4–27, 1990.
24. S. Ananthraman, Applying intelligent process optimization techniques in today's industry, *Ind. Comput.,* April 1995, pp. 40–44.
25. D. A. White and D. A. Sofge, *Handbook of Intelligent Control,* New York: Van Nostrand Reinhold, 1992.
26. S. Prabhu and D. Garg, A labelled object identification system using multi-level neural networks, *J. Inf. Sci.,* **3** (2): 111–126, 1995.
27. S. K. Ananthraman and D. P. Garg, Neurocontrol of cooperative dual robot manipulators, ASME Winter Annual Meeting, New Orleans, LA, **DSC-48**: 57–65, 1993.
28. S. M. Prabhu and D. P. Garg, Artificial neural networks in robotics: An overview, *J. Intelligent Robotic Syst.: Theory Appl.,* **15** (4): 333–365, 1996.
29. J. B. Bishop et al., Classification of movement patterns in patients with low back pain using an artificial neural network, in *Intelligent Engineering Systems through Artificial Neural Networks, ANNIE '96,* St. Louis, MO, November 10–13, 1996, pp. 699–704.
30. D. P. Garg, Adaptive control of nonlinear dynamic SCARA type of manipulators, *Robotica,* **9** (3): 319–326, 1991.
31. A. G. Barto, Reinforcement learning and adaptive critic methods, in D. A. White and D. A. Sofge (eds.), *Handbook of Intelligent Control,* New York: Van Nostrand Reinhold, 1992, pp. 469–491.
32. A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins, Learning and sequential decision making, Technical Report COINS TR 89-95, Department of Computer and Information Sciences, University of Massachusetts, 1989.
33. B. Widrow, D. Rumelhart, and M. Lehr, Neural networks: Applications in industry, business, and science, *Commun. ACM,* **37** (3): 93–105, 1994.
34. W. Staib and S. Ananthraman, Neural networks in control: A practical perspective gained from Intelligent Arc Furance™ operating experience, presented at 1994 World Congress on Neural Networks, San Diego, CA, June 4–9, 1994. This is available as a Technical Report from Neural Applications Corporation.

DEVENDRA P. GARG
Duke University

SANTOSH K. ANANTHRAMAN
Neural Applications Corporation

SAMEER M. PRABHU
CGN and Associates, Inc.