# NEURAL NET ARCHITECTURE

A neural network, as an artificial intelligence system, is capable of learning and computing. It consists of a group of processing units that are organized in a variety of architectures. The functional capacity of a neural network is largely determined by its architecture.

Artificial neural networks are conceptually inspired by the structure of biological systems, which consist of many interconnected neurons. While preserving the ability to perform the complex functions of biological systems, such as learning, generalization, error correction, information reconstruction, and pattern analysis, neural networks use simplified approaches to tackle those same problems, as reflected in their architecture.

A neural network can be characterized at two complementary levels: (1) architecture, by which the arrangements of units and the links among units are described, and (2) algorithm, by which the weights of links, as a function of learning, are modified so that the designated computation upon various inputs can be implemented.

Among commonly used architectures of neural networks are (1) the perceptron, (2) the multilayer feed-forward network, (3) the recurrent network, and (4) the radial basis function network.

Within a neural network, the links among units are locally stored as inherent rules, either explicitly or implicitly, when they are expressed analytically. Each unit alone has certain *simple* properties, but when interacting with each other, such as cooperating and competing, a neural network as an entity is able to complete many *complex* computational tasks.

A general architecture for neural networks is shown in Fig. 1. The processing within a neural network may be viewed as a functional mapping from input space to output space. In principle, a unit in a neural network can be represented using a mathematical function, and the weights associated with the unit can be represented in forms of coefficients of that function. The functional aggregation among different units, which creates the mapping from input to output space, is determined through both algorithm and architecture of a neural network.

## PERCEPTRON

The perceptron represents a type of neural network that consists of only the most basic processing units. As shown in Fig. 2, the architecture of a perceptron neural network includes several input units, one output unit, and *no* hidden units. Input units directly connect to the output unit through a set of weights, $W = (w_0, w_1, w_3, \ldots, w_n)$, which are usually determined through learning. The output unit is characterized by an activation function—either a step function or a linear function.

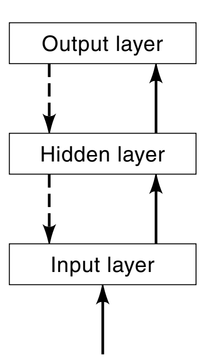


**Figure 1.** General neural-network architecture.

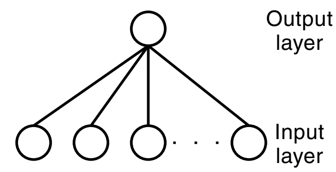


**Figure 2.** Perceptron architecture.

At the learning stage, the input units receive an input pattern vector $\boldsymbol{X} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$. A desired output $d$ is given to guide the learning. As a forward calculation, an inner product of $\boldsymbol{X}$ and $\boldsymbol{W}$ is computed and projected to the output unit. The output of a perceptron is defined as the results of the activity function of the output unit. The functional output for a perceptron $y$ is

$$y = f\left(\sum_{i=1}^{n}(w_i^* x_i) + \theta\right) \tag{1}$$

where $f$ is the activation function for the output unit and can be either a linear function or a step function. The input $x_i$ can be either a continuous analog value or a binary value. The weight $w_i$ is a continuous variable, and can take either a negative or positive value. The bias $\theta$ is a constant.

A learning algorithm automatically adjusts the weights $W$ so that the output $y$, as a function of input $\boldsymbol{X}$, will approach as closely as possible the respective desired output $d$. Error $E$, derived as a difference between $d$ and $y$, is minimized during the learning process.

The perceptron is functionally limited by its simple architecture of a single layer and a simple activation function. It can only be applied to classify those inputs that are linearly separable. In principle, if the inputs are not linearly separable (see Fig. 2), the learning of a perceptron can never reach a point where all input vectors are classified properly, but will converge to a linear least-squares fit.

As an example, a 2-input exclusive OR (XOR) problem is *nonlinearly separable.* It can be stated thus: a single output is on ($y = 1$) only if one or the other of the two inputs is on ($x_1 = 1$ and $x_2 = 0$; or $x_1 = 0$ and $x_2 = 1$), but not when neither or both inputs are on ($x_1 = 0$ and $x_2 = 0$; or $x_1 = 1$ and $x_2 = 1$). The output of the XOR problem contains two categories: *1* and *0*. No single straight line can separate these input patterns into the correct *1* and *0* categorizations (Fig. 3), thus, no single perceptron is able to implement the XOR problem.

The limitations of the perceptron can be overcome, to an extent, by other neural networks that are supported by more sophisticated architectures.

## MULTILAYER NETWORK

A multilayer neural network contains one or more hidden layers, in addition to input and output layers. The basic architec-

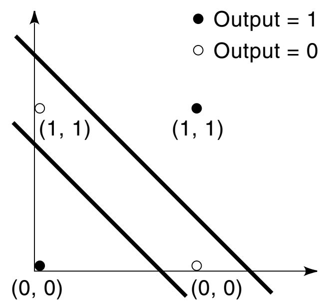


**Figure 3.** Nonlinearly separable.

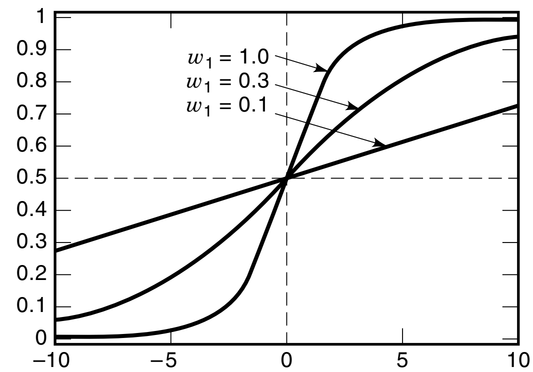


**Figure 5.** Sigmoidal function.

ture of a 3-layer neural network is illustrated in Fig. 4. Within this type of neural network, every unit on one layer connects to all units on the neighboring layer. The connection weights, which are associated with each unit, are all adjustable. The input layer first introduces external signals to the neural network and then projects these signals to the hidden layer(s). Each unit in a hidden layer has its own activation function (usually of sigmodal type). The hidden layer transforms the received signals through the associated activation functions and carries the resultant signals to the output layers. The feed-forward transformation and weighting operations play a central role in constructing a complex functional relationship between the inputs and the outputs of the neural network. The output layer combines the results of functional transformation on the hidden layer and generates a dependent outcome for the neural network.

The feed-forward calculation for a multilayer network may be described analytically. Starting from the first hidden layer ($k = 2$), the transformation between input and output is as follows: If a sigmoid type of activation function is used, then

$$O_j^k = \frac{1}{1 - e^{-\sum_m (W_{ij}^k * x_i^{k-1} + \theta^{k-1})}} \quad k - 2, 3, \ldots, n \tag{2}$$

where $O_j^k$ is the output from unit $j$ in layer $k$; $x_i^{k-1}$ is the output from unit $i$ in layer $k - 1$; $w_{ij}^k$ is the weight from unit $i$ in layer $k - 1$ to unit $j$ in layer $k$; and $\theta^{k-1}$ is the bias.

Figure 5 illustrates a 1-dimension sigmoidal function, as well as how the shape of the sigmoid function can be changed with the weights (coefficients in the equation). Therefore, by adjusting the weights of a network, the functional transformation in a network may be changed.

During learning, a neural network takes sample inputs, for which the corresponding outputs are known. Then, the responses of each output unit in the network are compared with the known outputs. Error signals associated with the output units are computed and backpropagated through the network. The backpropagated errors are used as indications for changing the connection weights for hidden layer(s). The weights are adjusted in a direction that minimizes the error. This process is the well-known backpropagation, discussed in detail in the algorithm section.

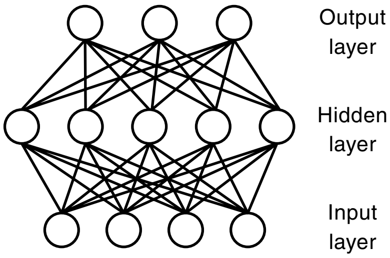


**Figure 4.** Multilayer neural-network architecture.

Multilayer networks can become complex systems through learning. The following describes how a network, which contains a single 2-unit hidden layer, learns to solve the XOR problem. Recall that a perceptron is unable to solve the nonlinearly separable problem.

The learning samples are (a) (0,0,0), (b) (0,1,1), (c) (1,0,1), and (d) (1,1,0), where the order in the parenthesis is (input 1, input 2, desired output). For hidden unit 1, we have

$$h_0(x_1, x_2) = \frac{1}{1 - e^{-(w_{00} * x_1 + w_{10} * x_2 + w_{20})}} \tag{3}$$

where $x_1$ and $x_2$ are the inputs, $w_{00}$, $w_{10}$ and $w_{20}$ are the weights from the input layer to the first hidden unit, and $w_{20}$ is the weight for bias input which has constant value 1.

For hidden unit 2, we have

$$h_1(x_1, x_2) = \frac{1}{1 - e^{-(w_{01} * x_1 + w_{11} * x_2 + w_{21})}} \tag{4}$$

where $w_{01}$, $w_{11}$, and $w_{21}$ are the weights from the input layer to the second hidden unit. The 2-dimensional surface plots shown in Fig. 6 and Fig. 7 illustrate the individual contributions of each hidden node.

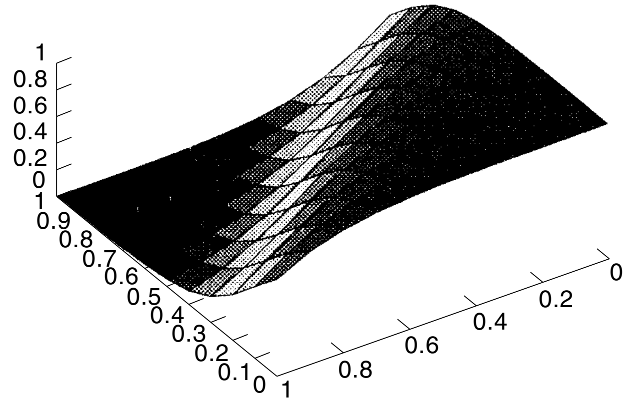


**Figure 6.** Output surface from first hidden unit.

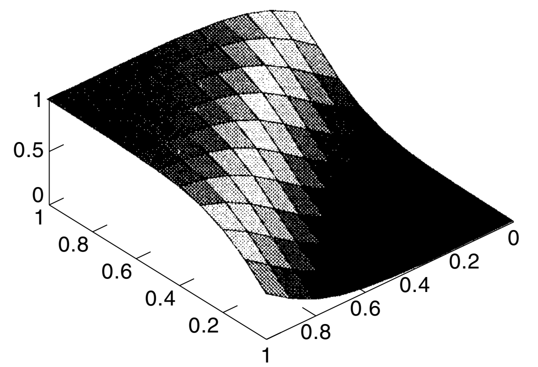


**Figure 7.** Output surface from second hidden unit.

Figures 6 and 7 show that after the learning, the first hidden unit fits three sample points. (1, 1, 0), (1, 0, 1), and (0, 1, 1). The second hidden unit also fits three sample points. (0, 0, 0), (1, 0, 1), and (0, 1, 1). Note that for each hidden unit, one of the samples is incorrectly classified; fortunately, the missed point is picked up by the other hidden unit. For the output unit, the two surfaces built by the two hidden units are weighted and summed. Therefore, the final output surface fits all four sample points. From a network architecture point of view, we can see that in order to solve the XOR problem, two hidden units are needed for the hidden layer—each hidden unit constructs a surface to solve part of the nonlinearly separable problem. The final outputs are determined by a combination of functions from the two hidden units, or a weighted sum of the two corresponding surfaces (Fig. 8).

The above figure shows that the network interpolates between the points during learning. Therefore, this architecture provides potential for generalization, which is one of the essential aspects of neural networks.

**The Necessary Number of Hidden Layers and Hidden Units.** In search of a solution for the XOR problem, we may have noticed that the neural network architecture, such as the number of units in a layer and the number of layers, plays a decisive role. The question arises, how many units and how many layers are necessary for a neural network? Up to now, we have an answer for the second but not the first question. It has been proven theoretically that, at most, *two* hidden layers are necessary for a network to approximate a particular set of functions to a given accuracy. It has also been proven that only *one* hidden layer is sufficient to approximate any continuous function. However, the necessary number of hidden units in each hidden layer is not known in general. Thus, the number of hidden units in each hidden layer is chosen experimentally.

**Generalization.** One of the reasons for much of the excitement about neural networks is their ability to extend learned knowledge into solving similar but not pre-exposed problems, the so-called generalization property. After learning a number of samples, a neural network can often establish a complete relationship that interpolates and extrapolates from the learned samples. If the network generates correct outputs with high probability to input patterns that were not included in the learned set, it is said that generalization has taken place. In learning of the above XOR problem, the network gives output values not only at (0, 0), (0, 1), (1, 0), (1, 1), the sample inputs, but also at any other inputs. For example, at (0.5,0.5), the network generates the output value .7832. More often than not, there are an almost infinite number of possible patterns of generalization. It should be pointed out, however, that when the architecture of a neural network becomes too complicated, (too many weights), poor generalization tends to occur, analogous to curve-fitting where too many free parameters may result in over fitting.

## RECURRENT NETWORK

The term recurrent network refers to a network that has direct or indirect links from units to themselves or from units to the units in previous layers. This type of neural network architecture makes recurrent networks capable of representing temple information (Fig. 9), which allows time-variant dynamic systems to be modeled. A recurrent network can restore and retrieve associated information in a flexible and time-dependent way.

In Figure 9, there are two kinds of recurrent links: individual feedback links, and links that connect to preceding hidden layer(s) and the input layer. In a recurrent network, any number or combination of recurrent links may be used. Different recurrent links may represent various internal functionalities of a neural network. Each type of recurrent link may be better suited to solve one rather than the other problems. A general expression for a recurrent network is

$$\boldsymbol{y}(t+1) = f_{net}(\boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{x}(t+1)) \tag{5}$$

where $\boldsymbol{y}(t+1)$ and $\boldsymbol{y}(t)$ represent the outputs at time $t+1$ and $t$, respectively, and $\boldsymbol{x}(t+1)$ and $\boldsymbol{x}(t)$ represent the inputs at time $t+1$ and $t$.

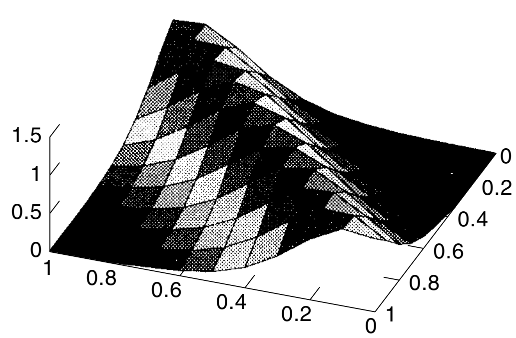


**Figure 8.** Output surface from the output unit.

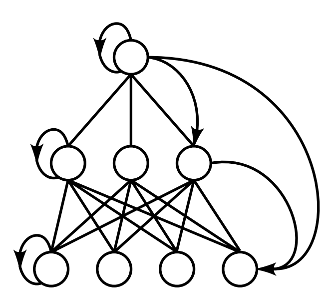

**Figure 9.** Recurrent network.

From the perspective of processing systems, multilayer networks can be characterized as static nonlinear functions, while recurrent networks can be characterized as nonlinear *dynamic* feedback functions. Recurrent networks can include internal states, and are built with an architecture that is good for applications for time-sensitive problems, such as,

1. Time sequence recognition, for which the network needs to produce a particular output when a specific input sequence is presented.
2. Time series prediction, for which the network needs to generate the rest of a sequence when it is presented with only part of the sequence.
3. Dynamic system modeling, for which the network needs to function as a model for a time-dependent system.

Recurrent networks, like multilayer networks, can learn through the presentation of samples, using the backpropogation algorithm.

## RADIAL BASIS FUNCTION NETWORK

Radial basis function networks have only one hidden layer and use radial basis functions as activation functions for the hidden layer. A radial basis function has one center and the functional response decreases with distance from the center. The radial basis function network is described as a specific architecture because of its localization property. By localization, it means that adjustment of an activation function for one of the hidden units only has effect on the region near its center. This regional property makes learning easier and faster for certain kinds of problems. J. Moody and Darken first proposed a network architecture that employs the basic concept of radial basis functions.

Most radial basis function networks use Gaussian functions as activation functions for hidden layers. Fig. 10 shows a one dimensional Gaussian function. Eq. (6) is the analytical expression for the Gaussian function.

$$y = \exp\left(\frac{-(x-\mu)^2}{\sigma^2}\right) \tag{6}$$

In the above equation, $\mu$ is the center of the Gaussian function, and $x$ is the input variable. If the input is away from the center $\mu$, the output $y$ will be close to zero. Therefore, when this kind function is used as an activation functions for hidden units, the corresponding units will only be activated when the inputs are within a certain neighborhood of the center. This is why localization occurs.

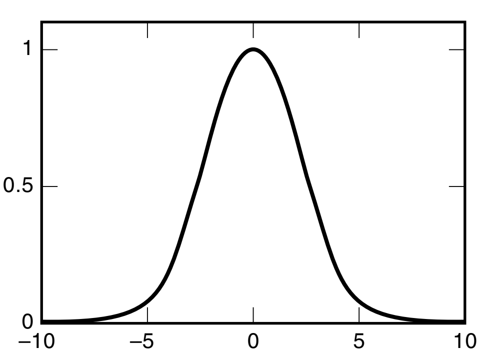


**Figure 10.** Gaussian function.

Each hidden unit constructs a localized bumplike function that is nonzero only within a small region around the center. The output units sum up the weighted bumplike activation functions in hidden layers and normalize the result to generate a smooth output. The centers of the Gaussian functions are chosen (usually randomly) before learning. During the learning stage, the width and the height of the Gaussian functions are adjusted by changing the weights. The general forward calculation for the radial basis function network is as follows:

From input layer to hidden node $j$

$$h_j(x) = \frac{\exp(-\boldsymbol{w}_j * (\boldsymbol{x}-\boldsymbol{\mu}_j)^2)}{\sum_k \exp(-(-\boldsymbol{w}_k * (\boldsymbol{x}-\boldsymbol{\mu}_k)^2)} \tag{7}$$

From hidden layer to output

$$y = \sum_j w_j * h_j(x) \tag{8}$$

where $\boldsymbol{w}_j$ is the weight vector from the input layer to hidden node $j$; $\boldsymbol{\mu}_j$ is the center vector for hidden node $j$; and $w_j$ is the weight from hidden node $j$ to the output node. Here hidden nodes use normalized Gaussian activation functions, and $\Sigma_j h_j(x) = 1$. Using normalized Gaussian activation usually improves the network's generalization.

## NEURAL NETWORKS FOR CONTROL

The primary objective of a controller is to generate appropriate signals for a plant so that desired outputs can be produced. Many types of neural networks have been considered for control. The main advantage of using a neural network controller is its adaptability to unforeseen situations. There are two main learning schemes for a neural network controller: (1) off-line learning for direct inverse control, and (2) online learning for the control of dynamic systems.

### Off-Line Learning for Inverse Control

In inverse control, a neural network functions as an inverse model of the plant. When a desired output for the plant is presented, the neural network, acting as a controller, produces a correct control signal. This control signal drives the plant to generate the desired output. For training a neural controller, sample data need to be collected when the plant operates independently (Eq. 9).

$$y = g(u) \tag{9}$$

Here, $u$ represents a possible action that is projected to the plant $g(.)$, and $y$ is the corresponding output produced by the plant. As to the controller, its function can be seen as an inverse problem. Thus, the input and the output of the plant are used as target output and input for the controller, respec-

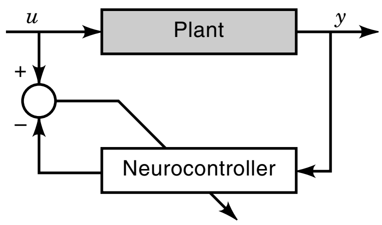


**Figure 11.** Inverse control training.

tively, at the learning stage (Fig. 11). Specifically, a set of sample data for training the controller should be ($y$, $u$), where $y$ is the input and $u$ is the target output. The neural network architecture for a controller can be a multilayer feed-forward network, recurrent network, or radial basis function network. After learning, the controller is connected to the plant and serves to control the plant (Fig. 12).

If $g(.)$ and $f(.)$ represent an unknown plant and the neural controller, respectively, then the inverse control process may be described as

$$u = f(x) \tag{10}$$

$$y = g(u) \tag{11}$$

where $x$ is the desired output for the plant, $u$ is the control signal generated by the neural controller, and $y$ is the controlled output from the plant. Since $f(.) \approx g^{-1}(.)$ after the controller is trained, the desired output from the plant is obtained by the neural controller.

$$y = g(u) = g[f(x)] \approx g[g^{-1}(x)] = x \tag{12}$$

This off-line learning for inverse control works well only for a static plant (that is, the input-output relationship does not vary with time). For time-varying dynamic plants, and for the plants without an inverse, the controller cannot be set up through the learning described above, no matter what neural network architecture is used. Under such circumstances, one solution is to train the controller with the plant separately, and then adjust the controller to adapt to any temporal changes of the plant. This process is called "backpropagation through plant."

### On-Line Learning for the Control of Dynamic Systems

In the control of dynamic systems, the learning of a neural network controller includes backpropagation through an identification neural network that acts as a model of the plant. This neural network control architecture originated from

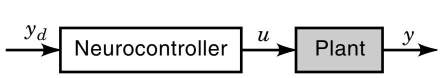


**Figure 12.** Inverse control.

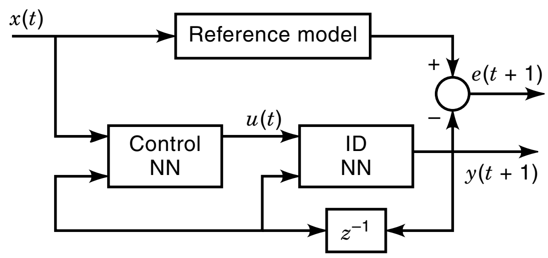


**Figure 13.** On-line learning for the control of dynamic systems. ID NN represents identification neural network. Control NN represents controller network.

work presented by Narendra, Psaltis, and Lightbody (Fig. 13). In addition to the neural controller, an identification neural network is introduced to model the plant. The identification neural network is trained first separately. After training the neural controller, the identification network is replaced by the real plant.

When training the neural controller, the neural network controller generates a control signal $u(t)$. Instead of being sent to the real plant, the control signal is fed to the identification neural network. The learning of the neural network controller cannot be carried out directly, since there is no desired control signal $u(t)$, that is, the inverse of the plant's desired output. This problem is solved with help of the identification neural network. The identification network backpropagates the output error to its input end. Here, the output error means a difference between the outputs of the identification network and of the reference model. With the backpropagated error as the correction signal, the learning of the neural controller can be carried out. After learning, the neural network controller is connected to the real plant.

The whole learning process is illustrated in Fig. 13. Here, the neural network controller receives not only external inputs, but also the inputs from the feedback of the plant. The learning procedures are employed such that the controller approximates a control function of the inputs. With the control function, the controller is able to generate the desired control for the plant. The outputs of the controller can be expressed as a function of the external input $x(t)$ and the feedback of the plant $y(t-1)$

$$u(t) = f[x(t), y(t-1)] \tag{13}$$

where $x(t) = [x(t), x(t-1), \ldots]^T$, and $y(t) = [y(t), y(t-1), \ldots]^T$. At the learning stage, error backpropagation is obtained by calculating the Jacobian of the identification network, as described in Eq. (14). For a cost function $E$ = (plant output − desired output)$^2$, its gradients for error propagation are derived with respect to weight $\boldsymbol{w}$ of the neural network controller.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial u}\frac{\partial u}{\partial w} + \left(\frac{\partial E}{\partial u}\frac{\partial u}{\partial y_{t-1}} + \frac{\partial E}{\partial y_{t-1}}\right)\frac{\partial y_{t-1}}{\partial w} \tag{14}$$

Then, the weights are adjusted as

$$\Delta w = -\eta^* e^* \frac{\partial E}{\partial w}{}^* \boldsymbol{X} \tag{15}$$

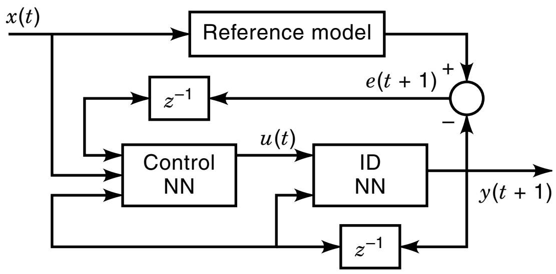


**Figure 14.** On-line learning for the control of dynamic systems with error feedback.

where $\eta$ is learning rate, $e$ is the difference between the reference model and the plant, and $\mathbf{X}$ is the input vector.

After the learning stage, the neural network controller supplies a control law. In principle, a neural network is able to approximate any arbitrary nonlinear functions. Thus, use of neural network provides an useful mean to solve an important problem—nonlinear control. Also, since all the parameters for the neural network controller, as well as for the neural network identification model, are obtained based on learning through samples, the corresponding control function may include some mathematically untraceable properties of the plant.

Due to the flexibility of neural network architectures, different connection schemes can be applied to the on-line neural network control. One alternative is to add error feedback to the controller to improve its adaptability. Using such an architecture, a nonlinear gain scheduled controller can be formed with specialization of a nonlinear continuous gain surface. Fig. 14 shows a neural network architecture for control with error feedback. Here, the control function of the neural network is no longer a fixed mapping from $x(t)$ to $u(t)$ for each state $y(t-1)$, but instead is the learned *slopes,* or the *gain,* referring to the feedback error $e(t)$. This gain is a continuous nonlinear function of the external input $x(t)$ and the state feedback $y(t-1)$. It should be pointed out that in order to use feedback error signals, this neural network controller needs to be recurrent at the output units (Fig. 15). The reason for using this architecture is that the output of the neural controller is a combination of old control signals, which are carried by the recurrent links, and a correction referring to the error $e(t)$. The other units of the neural network can be either feed forward or recurrent. If we denote the weight for the output units' recurrent link as $w_b$, then the output from the recurrent link is $w_b u(t-1)$. Eq. (16) describes the neural network with error feedback:

$$u(t) = w_b u(t-1) + f(x(t), y(t), e(t)) \tag{16}$$

where $f(.)$ is a nonlinear mapping function of the neural network when the recurrent link is not included, and $e(t) = [e(t), e(t-1), \ldots]^T$ is the feedback error based on the difference between the output of the plant and of the reference model. The main difference between this and other controllers is its inclusion of feedback error for control, which makes this feedback controller error driven. As long as the error exists, the control signal is adjusted so that the controller can, in principle, adapt to dynamic environments that were not encountered at the learning stage, such as varying physical properties of the plant.

The introduction of the neural network into control functions promises a useful approach to overcome some control problems. While providing a generic model for the broad class of systems considered in control theory, neural network control is specifically suitable in dealing with unknown dynamic systems. However, the architectures of the applied neural network need to be configured individually, depending on the details of the control problem. Meanwhile, on-line learning still confronts us with potential instability. The solutions to many of the remaining problems may rely on deeper understanding of the relationships between neural network architectures and learning algorithms.

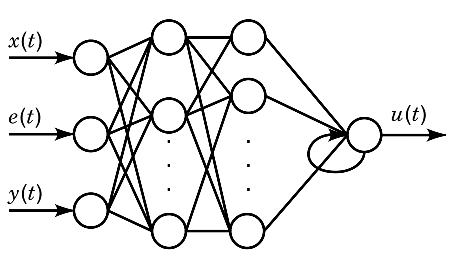


**Figure 15.** An example of neural network controller with a recurrent output unit.

## BIBLIOGRAPHY


1. C. Bishop, Improving the generalization properties of radial basis function neural networks, *Neural Computat.,* **3** (4): 579–588, 1991.
2. D. S. Broomhead and D. Lowe, Multivariate functional interpolation and adaptive networks. *Complex Syst.,* **2** (3): 321–355, 1988.
3. V. J. Bucek, *Control Systems: Continuous and Discrete,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
4. S. Chen, C. F. N. Cowan, and P. M. Grant, Orthogonal least squares learning for radial basis function networks. *IEEE Trans. Neural Netw.,* **2**: 302–309, 1991.
5. J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation,* Reading, MA: Addison-Wesley, 1991.
6. R. A. Jacobs and M. I. Jordan, A competitive modular connectionist architecture, in *Advances in Neural Information Processing Systems 3,* San Mateo, CA: Morgan Kaufmann, 1991, pp. 767–773.
7. R. Jang, C.-T. Sun, and E. Mizutani, *Neural-Fuzzy and Soft Computing,* Upper Saddle River, NJ: Prentice-Hall, 1997.
8. R. D. Jones et al., Function approximation and time series prediction with neural networks. *Proc. Int. Joint Conf. on Neural Netw.,* San Diego, CA, June 17–21, 1990, pp. 649–665.
9. G. Lightbody, Q. H. Wu, and G. W. Irwin, Control applications for feed networks, Chap. 4, in K. Warwich, G. W. Irwin and K. J. Hunt (eds.), *Neural Networks for Control Systems,* London: Peregrinus, 1992.
10. K. Liu, R. Tokar, and B. McVey, An integrated architecture of adaptive neural network control for dynamic systems, in *Advances in Neural Information Processing Systems 7,* San Mateo, CA: Morgan Kaufmann, 1991, pp. 1031–1038.
11. T. Miller, R. S. Sutton, and P. J. Werbos, *Neural Networks for Control,* Cambridge, MA: MIT Press, 1991.
12. J. Moody, The effective number of parameters: An analysis of generalisation and regularisation in nonlinear learning systems,

in J. E. Moody, S. J. Hanson, and R. P. Lippmann (eds.), *Neural Information Processing Systems 4,* San Mateo, CA: Morgan Kaufmann, 1992, pp. 847–854.

13. J. Moody and C. Darken, Learning with localized receptive fields, *Proc. 1988 Connectionist Models Summer School,* San Mateo, CA, 1988.
14. J. Moody and C. Darken, Fast learning in networks of locally-turned processing units, *Neural Computat.,* **1** (2): 281–294, 1989.
15. K. S. Narendra and K. Parthasarathy, Gradient methods for the optimization of dynamical systems containing neural networks, *IEEE Trans. Neural Netw.* **2**: 252–262, 1991.
16. T. Poggio and F. Girosi, Regularization algorithms for learning that are equivalent to multilayer networks, *Science,* **247**: 978–982, 1990.
17. D. Psaltis, A. Sideris, and A. Yamamura, Neural controllers, *Proc. of 1st Int. Conf. on Neural Networks,* Vol. 4, San Diego, CA, 1987, pp. 551–558.
18. G. V. Puskorius and L. A. Feldkamp, Recurrent network training with the decoupled extended kalman filter algorithm, in *Science of Artificial Neural Networks,* April 20–24, Orlando, FL, 1992.
19. K. J. Hunt et al., Neural networks for control systems—A survey, *Automatica,* **28** (6): 1083–1112, 1992.
20. W. H. Schiffmann and H. W. Geffers, Adaptive control of dynamic systems by back propagation networks, *Neural Networks,* **6** (4): 517–524, 1993.
21. S. Singhal and L. Wu, Training multilayer perceptrons with the extended Kalman algorithm, in D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 1,* San Mateo, CA: Morgan Kaufmann, 1989, pp. 133–140.
22. K. Stokbro, D. K. Umberger, and J. A. Hertz, Exploiting neurals with localized receptive fields to learn chaos, *Complex Syst.,* **4**: 603–622, 1990.
23. R. Zbikowski and P. J. Gawthrop, A survey of neural networks for control, Chap. 3, in K. Warwich, G. W. Irwin, and K. J. Hunt (eds.), *Neural Networks for Control and Systems,* London: Peregrinus, 1992.

KELLY LIU
The Mathworks Inc.

**NEURAL NET ARCHITECTURE.** See DISPATCHING.

**NEURAL NETS, ART.** See ART NEURAL NETS.