CARS&FOF 2000

APPLICATION OF ARTIFICIAL NEURAL NETWORKS IN THE DESIGN OF CONTROL SYSTEMS[*]

Hong Helena Mu Y. P. Kakad B. G. Sherlock

Department of Electrical and Computer Engineering
University of North Carolina at Charlotte
9201 University City Blvd, Charlotte, NC 28223, USA

ABSTRACT

The paper develops important fundamental steps in applying artficial neural networks in the design of intelligent control systems. Different architectures including single layered and multi layered of neural networks are examined for controls applications. The importance of different learning algorithms for both linear and nonlinear neural networks is discussed. The problem of generalization of the neural networks in control systems together with some possible solutions are also included.

INTRODUCTION

In recent years, the field of intelligent controls has become increasingly important due to the immense development in computing speed, power, and affordability. Neural network based control system design has become an important aspect of intelligent control since it is effective when mathematical models are inaccurate or not available. The method provides a distinctive computational paradigm to learn (or to adapt) linear or nonlinear mappings from a priori data and knowledge without fully understanding the physical behavior of the system. Thus the method does not rely on models based on traditional analytic differential equations and since the models are developed using computers, the control design produces controllers, that can be implemented online.
The paper examines both the nonlinear multi-layer feed-forward architecture and the linear single-layer architecture of Artificial Neural Networks for application in control system design. In the nonlinear multi-layer feed-forward case, the two major problems are the long training process and the poor generalization. To overcome these problems, a number of data analysis strategies before training and several improvement generalization techniques are used. Basically, neural design entails constructing a controller not only to meet the performance objectives like rise time, overshoot, settling time, and state error, but also to satisfy neural network standards such as a low number of local minima, higher learning speed, and better generalization capability under stable conditions.

ARCHITECTURES IN NEURAL NETWORKS

One of the major tasks in the design of a neural network is the selection of architecture and this choice depends on the nature of the problem. Inappropriate choice results into poor

[*] This work was supported in part by Office of Naval Research grant number N00014-99-1-0091

performance. The following are the commonly used neural network architectures for control system applications:

(a) *ADALINE Architecture* **(ADAptive LINear Element)***:* This architecture [1] is a single layer with linear output. It is the simplest intelligent self-learning system that can adapt itself to achieve a given modeling task. Figure 1 is a schematic diagram for such a network. Inputs connect directly to the outputs through a single layer of weights. It has a purely linear output unit; hence the network output y is a weighted linear combination of the inputs plus a constant term b_0 , i.e.

$$y = \sum_{i=1}^{n} w_i x_i + b_0 \tag{1}$$

ADALINE is simple in structure, therefore inexpensive. It can classify only data sets that are linearly separable. Equation (1) classifies an exact linear model with $n+1$ linear parameters, so we can employ least squares methods to minimize the error. However, most least-squares methods require extensive calculation, which is not possible in a physical system with simple components. Various methods [2] have been developed to reduce calculation and speed up convergence, and these are discussed in the next section.

(b) ***Feed-forward Neural Network Architecture:*** This architecture is the most popular structure in practice due to its non-parametric, non-linear mapping between input and output. Networks with this architecture are known as universal approximators, including multilayer feed-forward neural networks employing sigmoidal hidden unit activations. These networks can approximate not only an unknown function but also its derivative [3].

The Feed-forward neural networks include one or more layers of hidden units between the input and output layers. As its name suggests, the output of each node propagates from the input to the output side. All connections point from input towards output. See figure 2.

Multiple layers of neurons with nonlinear activation functions allow this type of neural network to learn nonlinear and linear relationships between input and output vectors. Each input has an appropriate weighting W. The sum of the weighted inputs and the bias B forms the input to the transfer function. Any differentiable activation function f may be used to generate the outputs. Three of the most commonly used activation functions are purelin $f(x) = x$, log-sigmoid $f(x) = \left(1+e^{-x}\right)^{-1}$, and tan-sigmoid $f(x) = \tanh(x/2) = (1-e^{-x})/(1+e^{-x})$.

Both the hyperbolic tangent (Tan-sigmoid) and logistic (Log-sigmoid) functions approximate the signum and step functions, respectively, and yet provide smooth, nonzero derivatives with respect to the input signals. These two activation functions called squashing functions since their outputs are squashed to the range [0, 1] or [-1,1]. They are also called sigmoidal functions because their S-shaped curves exhibit smoothness and asymptotic properties.

The activation function f_h of the hidden units have to be differentiable nonlinear functions [typically, f_h is the logistic function or hyperbolic tangent function f_h (net) = tanh (net)]. If f_h is linear, then one can always collapse the net to a single layer and thus lose the universal approximation/mapping capabilities. Each unit of the output layer is assumed to have the same activation function.

The node function for the output layer is a weighted sum with no squashing functions. This is equivalent to a situation in which the activation function is a purelin function, and output nodes of this type are often called linear nodes.

LEARNING ALGORITHM

In this section, a number of basic concepts necessary in the development of learning algorithms are presented and many of these algorithms are gradient-based. Learning in a neural network is normally accomplished through an adaptive procedure, known as a learning rule, whereby the weights of the network are incrementally adjusted so as to improve a predefined performance measure over time (figure 4). In other words, the process of learning is best viewed as an optimization process. More precisely, the learning process can be viewed as a "search" in a multidimensional parameter (weight) space for a solution, which gradually optimizes an objective (cost) function.

Gradient Based Methods

Gradient based method searches for minima by comparing values of the objective function $E(\theta)$ at different points. The basic idea is to evaluate the function at points around the current search point and then look for lower values. In other words, the primary concern is to minimize the objective function $E(\theta)$ in the adjustable space $\theta = [\theta_1 \theta_2, \dots, \theta_n]$ and to find a minimum point $\theta = \theta^*$. In general, a given function E depends on an adjustable parameter θ with a nonlinear objective function. $E(\theta) = f(\theta_1 \theta_2, \dots, \theta_n)$ is so complex that an iterative algorithm is often used to search the adjustable parameter space efficiently. In iterative descent methods, the next point θ_{new} is determined by a step down from the current point θ_{now} in a direction vector d given as:

$$\theta_{next} = \theta_{now} + \eta d \tag{2}$$

where η is some positive step size commonly referred to as the learning rate.
The principal differences between various descent algorithms lie in the first procedure for determining successive directions. Once the decision is reached, all algorithms call for movement to a (local) minimum point on the line determined by the current point θ_{now} and the direction d. That is, for the second procedure, the optimum step size can be determined by linear minimization [4] as:

$$\eta^* = \underset{\eta > 0}{\arg(\min(\phi(\eta))} \tag{3}$$

where

$$\phi(\eta) = E(\theta_{new} + \eta d). \tag{4}$$

When the straight downhill direction d is determined on the basis of the gradient (g) of an objective function E, such descent methods are called gradient based descent methods [4].
Gradient based descent method is not highly regarded among the optimization methods mainly because of its slow rate of convergence. The gradient never points to the global minimum except in the case where the error contours are spherical so many small steps are needed to arrive at the minimum.

Some common gradient methods are: steepest descent, Newton's method, Gauss-Newton method, and Levenberg-Marquardt method. The latter method has proven useful in the design of control systems and we adopt it here.

Levenberg-Marquardt Method

The Levenberg-Marquardt algorithm can handle ill-conditioned matrices well, like non-quadratic objective functions. Also, if the Hessian matrix is not positive definite, the Newton direction may point towards a local maximum, or a saddle point. The Hessian can be changed by adding a positive definite matrix λI to H in order to make H positive definite. Thus,

$$\theta_{next} = \theta_{now} - (H + \lambda I)^{-1} g \ , \tag{5}$$

where I is the identity matrix and H is the Hessian matrix which is given in terms of Jacobian matrix J as $H = J^T J$. Levenberg-Marquardt is the modification of the Gauss-Newton algorithm as

$$\theta_{next} = \theta_{now} - \left(J^T J\right)^{-1} J^T r \ = \theta_{now} - \left(J^T J + \lambda I\right)^{-1} J^T r \ . \tag{6}$$

The Levenberg-Marquardt algorithm performs initially small, but robust steps along the steepest descent direction, and switches to more efficient quadratic Gauss-Newton steps as the minimum is approached. This method combines the speed of Gauss-Newton with the everywhere-convergence of gradient descent, and appears to be fastest for training moderate-sized feedforward neural networks [3].

Forward-Propagation and Back-propagation

During training, a forward pass takes place. The network computes an output based on its current inputs. Each node i computes a weighted sum a_i of its inputs and passes this through a nonlinearity to obtain the node output y_i (fig.no. 3). The error between actual and desired network outputs is given by

$$E = \frac{1}{2} \sum_p \sum_i \left(d_{pi} - y_{pi}\right)^2 \tag{7}$$

where p indexes the patterns in the training set, i indexes the output nodes, and d_{pi} and y_{pi} are, respectively, the desired target and actual network output for the i th output node on the p th pattern.

The back-propagation algorithm is widely used used for training multi-layer neural networks, and is developed analytically here. The derivative of the error with respect to the weights is the sum of the individual pattern errors and is given as

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \frac{\partial E_p}{\partial w_{ij}} = \sum_{p,k} \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{\partial w_{ij}} \tag{8}$$

where the index k represents all output nodes. It is convenient to first calculate a value δ_i for each node i as

$$\delta_i = \frac{\partial E_p}{\partial a_i} = \sum_k \frac{\partial E_p}{\partial y_k} \frac{\partial y_k}{\partial a_i} \tag{9}$$

which measures the contribution of a_i to the error on the current pattern. For simplicity, pattern indexes p are omitted on y_i, a_i and other variables in the subsequent equations.
For output nodes, $\partial E_p / \partial a_k$, is obtained directly as

$$\delta = -\left(d_{pk} - y_{pk}\right) f' \qquad \text{(for output node).} \tag{10}$$

The first term in this equation is obtained from error equation, and the second term which is

$$\frac{\partial y_k}{\partial a_k} = f'(a_k) = f_k' \tag{11}$$

is just the slope of the node nonlinearity at its current value. For hidden nodes, δ_i is obtained indirectly as

$$\delta_i = \frac{\partial E_p}{\partial a_i} = \sum_k \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial a_i} \tag{12}$$

where the second factor is obtained by noting that if the node i connects directly to node k then $\partial a_k / \partial a_i = f_i' w_{ki}$, otherwise it is zero. Thus,

$$\delta_i = f_i' \sum_k w_k \delta_k \tag{13}$$

for hidden nodes.
In other words, δ_i is a weighted sum of the δ_k values of nodes k to which it has connections w_{ki}. The way the nodes are indexed, all delta values can be updated through the nodes in the reverse order. In layered networks, all delta values are first evaluated at the output nodes based on the current pattern errors, the hidden layer is then evaluated based on the output delta values, and so on backwards to the input layer.
Having obtained the node deltas, it is an easy step to find the partial derivatives $\partial E_p / \partial w_{ij}$ with respect to the weights. The second factor in (8) is $\partial a_k / \partial w_{ij}$ because a_k is a linear sum, this is zero if $k \neq i$; otherwise

$$\frac{\partial a_i}{\partial w_{ij}} = x_j \quad . \tag{14}$$

The derivative of pattern error E_p with respect to weight w_{ij} is then

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_i x_j \ . \tag{15}$$

First the derivatives of the network training error with respect to the weights are calculated. Then, a training algorithm is performed. This procedure is called back-propagation since the error signals are obtained sequentially from the output layer back to the input layer.

Weight Update Algorithm

The reason for updating the weights is to decrease the error. The weight update relationship is

$$\Delta w_{ij} = \eta \frac{\partial E_p}{\partial w_{ij}} = \eta (d_{pi} - y_{pi}) f_i' x_j \tag{16}$$

where the learning rate $\eta > 0$ is a small positive constant. Sometimes η is also called the step size parameter.

The Delta Rule is a weight update algorithm in the training of neural networks. The algorithm progresses sequentially layer by layer, updating weights as it goes. The update equation is provided by the gradient descent method as

$$\nabla w_{ij} = w_{ij}(k+1) - w_{ij}(k) = -\eta \frac{\partial E_p}{\partial w_{ij}} \tag{17}$$

$$\frac{\partial E_p}{\partial w_{ij}} = -\left(d_{ij} - y_{pi}\right)\frac{\partial y_{pi}}{\partial w_{ij}} \tag{18}$$

for linear output unit, where

$$y_{pi} = \sum_i w_{ij} x_i \tag{19}$$

and

$$\frac{\partial y_{pi}}{\partial w_{ij}} = x_i \tag{20}$$

so,

$$\nabla w_{ij} = w_{ij}(k+1) - w_{ij}(k) = \eta\left(d_{pi} - y_{pi}\right)x_i \tag{21}$$

The adaptation of those weights which connect the input units and the ith output unit is determined by the corresponding error $e_i = \frac{1}{2}\sum_p \left(d_{pi} - y_{pi}\right)^2$.

Training Data Analysis

A common complaint about back-propagation is the long training time. A typical training session may require thousands of iterations. Large networks with large training sets might take days or weeks to be trained. Two training data analysis methods, namely (1) normalizing training set and initializing weights, and (2) Principal Components Analysis (PCA) are intended to speed up the learning process.
The first method speeds up the training by choosing better solutions, by controlling the distribution of random initial weights and by avoiding sigmoid saturation problems that cause slow training.
The Principal Components Analysis (PCA) method can reduce the dimensionality of training data presented to a neural network without losing necessary information. It can speed up training significantly. One basis for selecting an initial weight distribution is to assume that the inputs have some statistical distribution and select the initial weight distribution so that the probability of saturating the node nonlinearity is small. Finding the optimal architecture of the neural network is as important as selecting an efficient learning algorithm. The total number of nodes also influences training speed. For a very large set of input variables with insufficient information, the network may not be able to estimate the relation very accurately. Using Principal Components Analysis (PCA) to analyze the data before applying to a neural network will help to overcome the above listed disadvantages.
PCA is a statistical technique that transforms a large set of correlated variables into a smaller set of uncorrelated variables containing most of the original information. It can be effectively used as part of the neural network training procedure.

Improving Generalization

The critical issue in developing a neural network is its potential for generalization. A network may be insufficiently complex and fail to fully detect the signal, leading to *underfitting*. A network that is too complex may fit the noise as well as the signal, leading to *overfitting*. Overfitting is especially dangerous because it can lead to predictions beyond the range of the training data. Overfitting can produce wild predictions even with noise-free data. Underfitting produces excessive bias in the outputs, whereas overfitting produces excessive variance. Overfitting can be remedied by using extra training data. When the amount of training data is fixed, three approaches to avoiding underfitting and overfitting, thereby getting good generalization, may be used: (1) Selection of the model so that the size of weights as well as the numbers of weights, hidden units and layers, are appropriate to the number of training cases; (2) adding small amounts of jitter (noise) to inputs during training to increase the effective number of training cases used; (3) performing cross-validation checks during training to determine when to stop.

CONCLUSION

The fundamentals of neural network based control system design are developed in this paper and are applied to intelligent control of the advanced flight propulsion control coupling (FPCC) aircraft. The results and applicability to the linearized lateral dynamics of the aircraft will be presented at the conference.

REFERENCES

1. S.I. Amari, N. Murata, K.R. Muller, M. Fincke, H.H. Yang (1997), "Asymptotic Statistical Theory of Overtraining and Cross-Validation", *IEEE Trans. Neural Networks*, 8(5), 985-993.
2. C.H. Dagli, M.Akay, O. Ersoy, B.R. Fernandez, A. Smith (1997), "Intelligent Engineering Systems Through Artificial Neural Networks", Vol. 7 of *Neural Networks Fuzzy Logic Data Mining Evolutionary Programming.*
3. H. Demuth and M. Beale (1997), *Neural Networks Toolbox Users Guide* , Mathworks.
4. L. Fu, *Neural Networks in Computer Intelligence* (1994), Prentice Hall.
5. M.T. Hagan, and M.B. Menhaj (1994), "Training Feedforward Networks with the Marquardt Algorithm", *IEEE Trans. Neural Networks*, 5(6), 989-993.

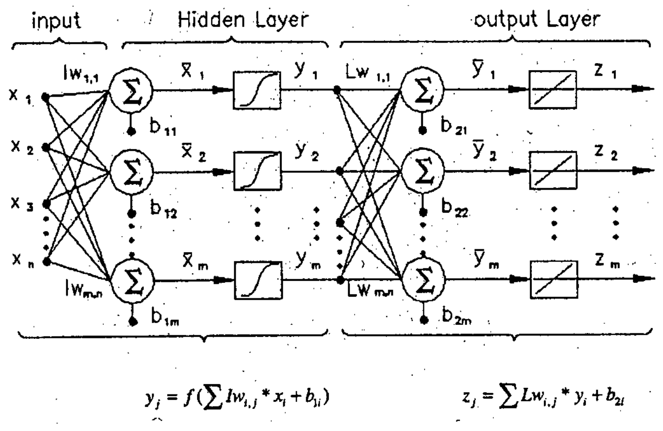

Figure 1. ADALINE Architecture

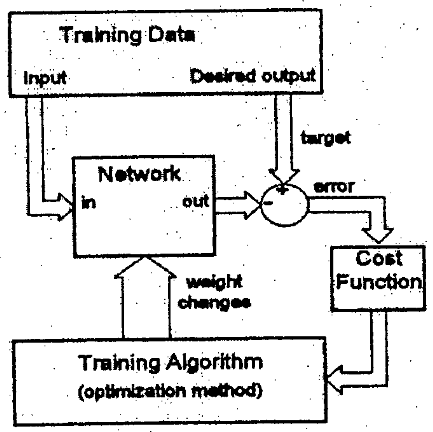

Figure 4. Learning process

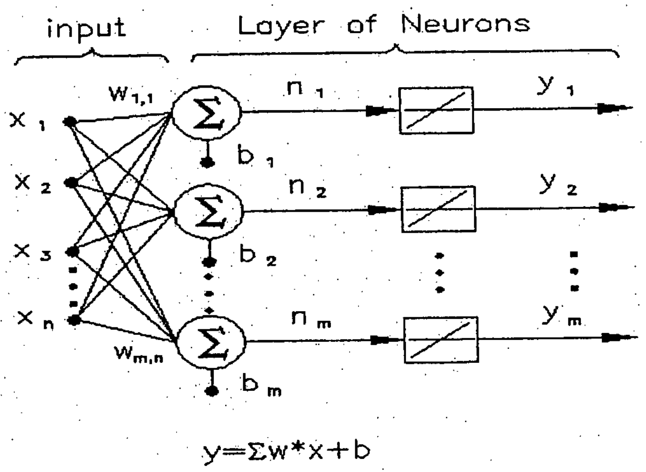

Figure 2. Feedforward neural network.

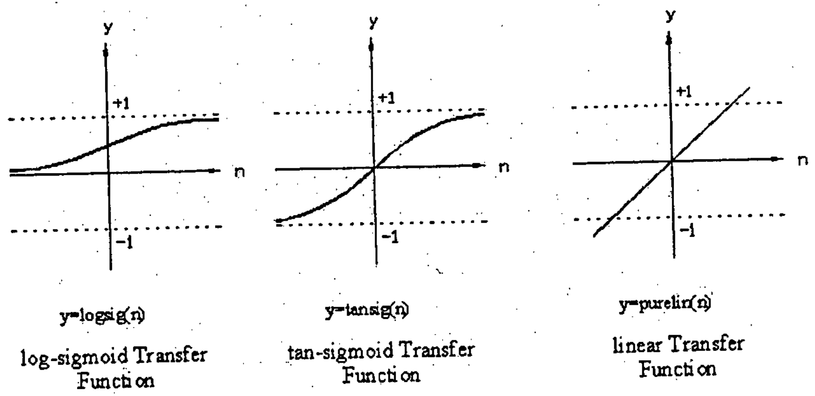

Figure 3. Three types of activation function