Memory Neuron Networks for Identification and Control of Dynamical Systems

P. S. Sastry, *Member, IEEE,* G. Santharam, and K. P. Unnikrishnan

***Abstract*—This paper discusses *Memory Neuron Networks* as models for identification and adaptive control of nonlinear dynamical systems. These are a class of recurrent networks obtained by adding trainable temporal elements to feed-forward networks that makes the output history-sensitive. By virtue of this capability, these networks can identify dynamical systems without having to be explicitly fed with past inputs and outputs. Thus, they can identify systems whose order is unknown or systems with unknown delay. It is argued that for satisfactory modeling of dynamical systems, neural networks should be endowed with such internal memory. The paper presents a preliminary analysis of the learning algorithm, providing theoretical justification for the identification method. Methods for adaptive control of nonlinear systems using these networks are presented. Through extensive simulations, these models are shown to be effective both for identification and model reference adaptive control of nonlinear systems.**

I. Introduction

There has been considerable interest in the past few years in exploring the applications of artificial neural networks (ANNs) for identification and adaptive control of dynamical systems [1], [2], [3], [4], [5]. In this paper we present a class of recurrent neural networks called Memory Neuron Networks [6] as general models for identification and control of nonlinear dynamical systems. These networks are obtained by adding some trainable temporal elements to feed-forward networks. The main attraction of these networks is that they have trainable internal memory and hence can directly model dynamical systems.

Identification of a system requires picking one of a class of functions (or models) so as to approximate the input-output behavior of the system in the "best" possible manner. In many situations, such as identification of dynamical systems, recognition of temporal patterns, etc., the output of the physical system to be modeled is a function of past inputs and outputs as well. In all such cases, the identification problem is complicated because the model being used (e.g., ANN) should have some internal memory. The output of a feed-forward ANN is a function of its current inputs only and hence it can model only memoryless transformations. In spite of this limitation, almost all attempts at using ANNs for identification and control rely on feed-forward nets [2], [3], [4], [5], [7], [8]. This is achieved by using the so called tapped delay line [9]. If the order of the system (or an upper bound on the order) is known, all the necessary past inputs and outputs of the system being modeled can be fed as explicit inputs to the network. Then the network can learn the memoryless transformation that captures the dependence of the output of the system on the specified past inputs and outputs. While, in principle, we can always feed a "sufficient" number of past values to the network, in practice, all reported applications assume that the exact order is known. Both from aesthetics and practicality, learning memoryless transformations in this manner may not lead to versatile dynamical models (see the discussion in [9] and Section V of this paper).

Recently, Narendra and Parthasarathy [3], [4] have shown that a rich class of models can be constructed by using ANNs and linear filters in cascade and/or feedback configurations. The ANN corresponds to the nonlinear part and the linear filter gives dynamics to the model. They have also worked out a method (called dynamic back propagation) to back-propagate errors through a linear dynamical system. Using simulations they showed that these models can identify some complicated nonlinear dynamical systems. While this is one elegant way to introduce dynamics into the model, it has a few drawbacks. One needs a fairly good knowledge of the structure of the system to decide what combination of memoryless transformations and linear filters is a good class of models to choose.

Now the natural question is: can one include dynamics directly into the network structure so that we can learn nonlinear dynamic systems without assuming much knowledge of the systems. For this, what we need are networks with some "internal memory" and learning algorithms for such recurrent networks. Due to the proven ability of feed-forward networks to model nonlinear systems (when explicitly fed with the necessary past history), it seems logical to explore recurrent networks that the closely related to multilayer feed-forward networks. Feed-forward networks with some memory in the form of external delay lines have been used successfully for recognition of speech signals [10].

In this paper we describe a recurrent network model with internal memory called the Memory Neuron Network (MNN) [6], [11]. Here each unit of neuron has, associated with it, a memory neuron whose single scalar output summarizes the history of past activations of that unit. These memory neurons, or more precisely the weights of connection into them, represent trainable dynamical elements of the model. Since the

Manuscript received February 1, 1993, revised August 12, 1993. This work was supported in part by the United States Agency for International Development (USAID) and Dept. Science and Technology, Govt. of India.

P. S. Sastry and G. Santharam are with the Department of Electrical Engineering, Indian Institute of Science, Bangalore, 560012, India.

K. P. Unnikrishnan is with the Computer Science Department, General Motors Research Laboratories, Warren, MI 48090-9055, USA, and the AI Laboratory, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9214801.

connections between a unit and its memory neuron involve feedback loops, the overall network is now a recurrent one.

Here we show, through extensive simulations, that this level of internal memory and trainable temporal elements are sufficient for identifying nonlinear dynamical systems. Our networks are fairly small and the learning algorithm (which is an approximation to gradient descent) is robust.

Thus far we have been talking only about identification of dynamical systems. One of the main reasons for identification is that the identified model can be used in deriving a controller for the system. Due to lack of a general theory, there are at present few classes of nonlinear models for which controllers can be designed using analytical techniques. This is one of the reasons for an increasing interest in using ANN for adaptive control. One of the approaches is to use ANN for identifying the nonlinear systems and then derive controller networks based on this. However, the virtual inscrutability of a ANN means we have obtained what should be termed as a nonparametric identifier [12]. Hence one needs to exploit a lot of knowledge regarding the system to construct such indirect controllers [3]. There are also a few general methods suggested for designing direct adaptive controllers based on Neural Networks. The main problem to be addressed is how does one get sufficient information to train the controller network [2]. We show that MNN can be used for model reference adaptive control. We make use of a continually updated forward model of the plant as a channel of back propagation to train the controller net (a technique variously called differentiating the model [2] or forward and inverse modeling [13]). We illustrate the tracking abilities of the controller (when the reference input changes) and also the online adaptation possible with these models (when the characteristics of the plant change during operation).

The primary aim of this paper is to illustrate the utility of recurrent network models for identification and control. We do not discuss many details about other possible strategies for training the controller. However this strategy of back-propagating through a plant model is a fairly general purpose method [14]. We point out that this technique as used now with feed-forward ANN can lead to wrong results, specially when the current output of the plant depends on more than one of the past inputs. We explain one method of overcoming this problem.

The rest of this paper is organized as follows. Section II presents the architecture of a Memory Neuron Network along with the learning algorithm. In Section III we discuss the problem of identification. We present a number of examples to illustrate the capabilities of the network. We also discuss the asymptotic properties of the learning algorithm and provide a theoretical justification for the identification method. Section IV discusses the problem of model reference adaptive control using these networks and presents simulation results. In Section V we summarize the salient features of our approach and conclude the paper in Section VI.

II. Memory Neuron Networks

In this section we describe the structure of the network that we use and the associated learning algorithm. The network we

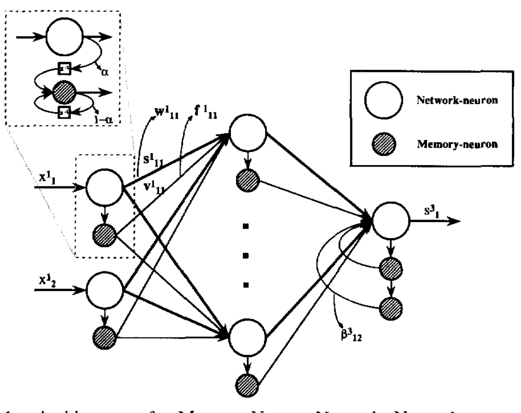


Fig. 1. Architecture of a Memory Neuron Network. Network neurons are shown as big open circles and memory neurons are shown as small shaded circles. Every network neuron, except those in the output layer, has one corresponding memory neuron. A network with m hidden layer nodes and n memory neurons per node in the output layer is referred to as a $m : n$ network. The expanded box shows the actual connections between a network neuron and a memory neuron. Memory neurons in the output layer feed onto their parent network neuron.

use is similar to the one described in [6]. The learning algorithm is also a slight modification of the one in [6] in order to make it conform better to a specific approximation of gradient descent for recurrent networks. It may be pointed out here that it is possible to use other incremental learning algorithms, e.g., ALOPEX [15], RTRL [16], with this network. We have kept the squared error criterion and the back propagation algorithm as this will clearly bring out the advantages of adding temporal elements to standard feed-forward networks.

A. Network Structure

The architecture of a Memory Neuron Network (MNN) is shown in Fig. 1. The structure is the same as a feed-forward ANN except for the memory neurons (shown by small filled circles in Fig. 1) attached to each unit in the network (shown by large open circles in Fig. 1). To distinguish these two types of units, we use the terms *network neuron* and *memory neuron*. As can be seen from Fig. 1, at each level of the network except the output level, each of the network neurons has exactly one memory neuron connected to it. The memory neuron takes its input from the corresponding network neuron and it also has a self feedback as shown in the inset in Fig. 1.[1] This leads to accumulation of past data of the network neuron in the memory neuron. All the network neurons and the memory neurons of each level send their outputs to the network neurons of the next level. In the output layer, each network neuron can have a cascade of memory neurons and each of them send their output to that network neuron in the output layer.[2] Fig. 1 shows a network with two input nodes, one output node and a single hidden layer.

[1]In Fig. 1, we show this connection between a network neuron and a memory neuron in only one expanded box. Otherwise we represent this connection by a single arrow from the network neuron to the memory neuron. In the rest of the paper we follow this convention to keep the figures simple though the actual connection is as shown in the expanded box in Fig. 1.

[2]The introduction of the memory neurons at the output layer is a modification to the structure in [6].

B. Dynamics of the Network

We use the following notation to describe the functioning of the network.

- L is the number of layers of the network with layer 1 as the input layer and layer L as the output layer.
- N_ℓ is the number of network neurons in layer ℓ.
- $x_j^\ell(k)$ is the net input to the j^{th} network neuron of layer ℓ at time k.
- $s_j^\ell(k)$ is the output of the j^{th} network neuron of layer ℓ at time k.
- $v_j^\ell(k)$ is the output of the memory neuron of the j^{th} network neuron in layer ℓ at time $k, 1 \le \ell < L$.
- $w_{ij}^\ell(k)$ is the weight of the connection from i^{th} network neuron of layer ℓ to j^{th} network neuron of layer $\ell+1$ at time k.
- $f_{ij}^\ell(k)$ is the weight of the connection from the memory neuron corresponding to the i^{th} network neuron of layer ℓ to the j^{th} network neuron of layer $\ell+1$ at time k.
- $\alpha_j^\ell(k)$ is the weight of the connection from j^{th} network neuron in layer ℓ to its corresponding memory neuron at time $k, 1 \le \ell < L$.
- $\alpha_{ij}^L(k)$ is the weight of the connection from the $(j-1)^{\text{th}}$ memory neuron to the j^{th} memory neuron of the i^{th} network neuron in the output layer at time k[3].
- $v_{ij}^L(k)$ is the output of the j^{th} memory neuron of the i^{th} network neuron in the output layer at time k.
- $\beta_{ij}^L(k)$ is the weight of the connection from the j^{th} memory neuron of the i^{th} network neuron to the i^{th} network neuron in the output layer at time k.
- M_j is the number of memory neurons associated with the j^{th} network neuron of the output layer.
- $g(.)$ is the activation function of the network neurons.

We shall refer to $\alpha_j^\ell, \alpha_{ij}^L$ and β_{ij}^L as memory coefficients.

The net input to the j^{th} network neuron of layer $\ell, 1 \le \ell < L$, at time k is given by

$$x_j^\ell(k) = \sum_{i=0}^{N_{\ell-1}} w_{ij}^{\ell-1}(k) s_i^{\ell-1}(k) + \sum_{i=1}^{N_{\ell-1}} f_{ij}^{\ell-1}(k) v_i^{\ell-1}(k) \quad (1)$$

In the above equation we assume that $s_0^\ell = 1$ for all ℓ and thus w_{0j}^ℓ is the bias for the j^{th} network neuron in layer $\ell+1$. This zeroth neuron at each level is only for notational convenience in dealing with the bias term and hence it will have no memory neuron. Thus v_0^ℓ does not exist and hence the summation index starts from 1 in the second sum in (1). The output of a network neuron is given by:

$$s_j^\ell(k) = g(x_j^\ell(k)), \quad 1 \le \ell \le L. \quad (2)$$

We make use of two different activation functions given by

$$g_1(x) = c1 \frac{1}{1+\exp(-k_1 x)}, \quad g_2(x) = c2 \frac{1-\exp(-k_2 x)}{1+\exp(-k_2 x)} \quad (3)$$

where g_1 is used for all the hidden nodes and g_2 is used for the output nodes. Here $c1, c2, k_1$ and k_2 are parameters of the activation function.

[3]Only the input layer contains more than one memory neuron for each network neuron.

For the units in the output layer, the net input is given by:

$$x_j^L(k) = \sum_{i=0}^{N_{L-1}} w_{ij}^{L-1}(k) s_i^{L-1}(k) + \sum_{i=1}^{N_{L-1}} f_{ij}^{L-1}(k) v_i^{L-1}(k) + \sum_{i=1}^{M_j} \beta_{ji}^L(k) v_{ji}^L(k) \quad (4)$$

The output of all the memory neurons except for those in the output layer, are derived by

$$v_j^\ell(k) = \alpha_j^\ell(k) s_j^\ell(k-1) + (1-\alpha_j^\ell(k)) v_j^\ell(k-1) \quad (5)$$

For memory neurons in the output layer,

$$v_{ij}^L(k) = \alpha_{ij}^L(k) v_{ij-1}^L(k-1) + (1-\alpha_{ij}^L(k)) v_{ij}^L(k-1) \quad (6)$$

where, by notation, we have $v_{i0}^L = s_i^L$. To ensure stability of the network dynamics, we impose the conditions: $0 \le \alpha_{ij}^L, \alpha_i^\ell, \beta_{ij}^\ell \le 1$.

With the above description of the network dynamics, it is easy to see the relationship between feed-forward networks and Memory Neuron Networks. Here the network neurons connected through weights, w_{ij}^ℓ constitute the feed forward part. Each of the memory neurons stores a combination of all the previous activations of the network neuron. The output of the memory neuron is obtained by passing the output of the network neuron through a first order filter. By keeping the memory coefficients between zero and one, we are ensuring the stability of this filter. These outputs are easily calculated locally, by remembering the values for one time step, using (5) and (6). Since the outputs of memory neurons contribute to the net input of the network neurons at the next level, the internal memory of the network plays a significant role in determining the output of the network at any time. The memory neurons at the output nodes allow for direct dependence of the current output of the network on its past outputs.

Though the network is a recurrent one, the manner of computing the output is very similar to that of a feed-forward net and we do not wait for the network to settle down to a stable state, etc. However, the output of the net at any given time is influenced by all the past inputs to the network [see (5) and (6)]. The degree of this influence will be determined by the magnitude of the memory coefficients (see [6] for a discussion of this point).

C. Learning Algorithm

In this section we describe the learning algorithm to be used for the Memory Neuron Network. As explained earlier, at each instant, we supply an input and calculate the output of the net using (1)–(6). Then we get a teaching signal and use that to calculate the error at the output layer and update all weights in the network. We use the usual squared error given by

$$e(k) = \sum_{j=1}^{N_L} (s_j^L(k) - y_j(k))^2 \quad (7)$$

where, $y_j(k)$ is the teaching signal for the j^{th} output node at time k.

We will be using a back propagation type algorithm. Thus, all we need are the derivatives of the error, $e(k)$, with respect to the weights in the network. Due to the presence of memory neurons, it is not easy to find exact partial derivatives through back propagation only at a single time step. Our strategy had been to stick to an approximation whereby we unfold the network in time by exactly one time step and then back-propagate the error. This means we can update all the weights at time k without needing anymore storage of the past activations of the nodes than is needed to implement (5) and (6).

Under this strategy, the final equations for updating the weights are given below. (These can be derived easily using the chain rule).

$$w_{ij}^{\ell}(k+1) = w_{ij}^{\ell}(k) - \eta e_j^{\ell+1}(k) s_i^{\ell}(k), \quad 1 \le \ell < L \quad (8)$$

when η is the step-size and

$$e_j^L(k) = (s_j^L(k) - y_j(k)) g'(x_j^L(k))$$
$$e_j^{\ell}(k) = g'(x_j^{\ell}(k)) \sum_{p=1}^{N_{\ell+1}} e_p^{\ell+1}(k) w_{jp}^{\ell}(k), \quad 1 \le \ell < L \quad (9)$$

The above is the standard back propagation of error without considering the memory neurons. $g'(.)$ is the derivative of the activation function of the network neuron and we need to use the appropriate function depending on the layer number (cf. (3)).

The updating of f is same as that of w except that we use the output of the corresponding memory neuron rather than the network neuron.

$$f_{ij}^{\ell}(k+1) = f_{ij}^{\ell}(k) - \eta e_j^{\ell+1}(k) v_i^{\ell}(k), \quad 1 \le \ell < L \quad (10)$$

The various memory coefficient are updated as given below.

$$\alpha_j^{\ell}(k+1) = \alpha_j^{\ell}(k) - \eta' \frac{\partial e}{\partial v_j^{\ell}}(k) \frac{\partial v_j^{\ell}}{\partial \alpha_j^{\ell}}(k), \quad 1 \le \ell < L \quad (11)$$

$$\alpha_{ij}^L(k+1) = \alpha_{ij}^L(k) - \eta' \frac{\partial e}{\partial v_{ij}^L}(k) \frac{\partial v_{ij}^L}{\partial \alpha_{ij}^L}(k), \quad (12)$$

$$\beta_{ij}^L(k+1) = \beta_{ij}^L(k) - \eta' e_i^L v_{ij}^L \quad (13)$$

where

$$\frac{\partial e}{\partial v_j^{\ell}}(k) = \sum_{s=1}^{N_{l+1}} f_{js}^{\ell}(k) e_s^{\ell+1}(k) \quad (14)$$

$$\frac{\partial v_j^{\ell}}{\partial \alpha_j^{\ell}}(k) = s_j^{\ell}(k-1) - v_j^{\ell}(k-1) \quad (15)$$

$$\frac{\partial e}{\partial v_{ij}^L}(k) = \beta_{ij}^L(k) e_i^L(k) \quad (16)$$

$$\frac{\partial v_{ij}^L}{\partial \alpha_{ij}^L}(k) = v_{ij-1}^L(k-1) - v_{ij}^L(k-1) \quad (17)$$

It may be noted that we are using two step-size parameters in the above equations—η' for the memory coefficients and η for the remaining weights. To ensure stability of the network, we project the memory coefficients back to the interval (0, 1), if after the above updating they are outside the interval.

III. MNN for Identification of Dynamical Systems

In this section we discuss the applications of Memory Neuron Networks for identification of nonlinear dynamical systems. We shall describe results of computer simulations of MNN identifying a variety of nonlinear plants.

For simplicity, we explain these ideas using a single input single output (SISO) plant. Denote by $u(k)$ and $y_p(k)$ the input and output of a SISO plant. Now consider a Memory Neuron Network with a single input node (i.e., one network neuron in the input layer), a single output node and some hidden nodes. Let the input node be fed $u(k)$ at time k, and let the output of the network at time k be $\hat{y}_p(k)$. We will use $y_p(k)$ as the teaching signal at time k. Now we have,

$$\hat{y}_p(k) = F(u(k), u(k-1), \ldots, \hat{y}_p(k-1), \hat{y}_p(k-2), \ldots) \quad (18)$$

Here F is the nonlinear transformation represented by the network. As explained in the previous section, $\hat{y}_p(k)$ depends on the previous inputs due to the memory neurons at input and hidden layers, and it depends on its own previous outputs due to the cascade of memory neurons at the output layer. The model represented by (18) is known as parallel identification model [3]. Thus, if we feed the current input to the plant as the sole input to the network and decide to use the output of the plant as the teaching signal, then MNN is a parallel identification model. This is in sharp contrast to the case of feed-forward net based techniques where one has to decide 1) how many previous inputs of the plant should be fed at the input of network and 2) how many past outputs of the network should be fed back to get a parallel identification model.

We get what is known as a series-parallel model for identification [3], if we let the current output of the model depend on the actual past outputs of the plant. In our case, we get a series-parallel model (for an SISO plant) by having a network with two input nodes to which we feed $u(k)$ and $y_p(k-1)$. This identification system is shown in Fig. 2. The single output of the net will be $\hat{y}_p(k)$. Now writing the output of this two input network as a function of all past inputs etc., simlar to (18) for the single input net, we get

$$\hat{y}_p(k) = F(u(k), u(k-1), y_p(k-1), \ldots, \hat{y}_p(k-1), \ldots) \quad (19)$$

It is easy to see that $\hat{y}_p$ will depend on the past inputs and outputs of the plant (and also the current input). The output of the network will also depend on its own past values. This dependence can be eliminated by removing memory neurons from the output layer. We stress once again that to get the series-parallel model, we do not need to know the order of the system. The network will automatically learn the relative weightage to be given to various past values and this weighted history is available through the memory neurons.

The series-parallel model is often found to be more useful for generating stable adaptive laws and we will be using it in this paper. To identify a m-input, p-output plant we will use a network with $m+p$ inputs and p outputs. This will be the case irrespective of the order of the system. We shall use the actual outputs of the plant at each instant as the teaching signals.

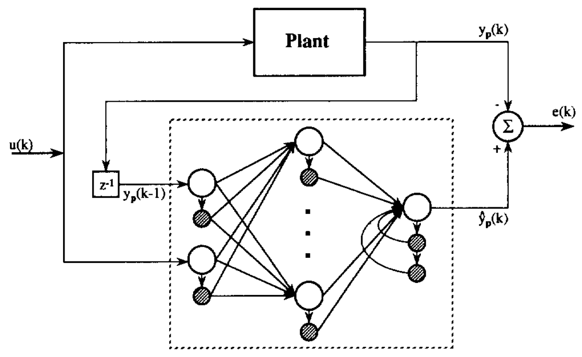


Fig. 2. Series parallel identification model with the Memory Neuron Network. The current input into the plant and the most recent output of the plant are fed into the network. The error $e(k)$ is used for learning the network parameters.

A. Analysis of the Identification Algorithm

Before we describe simulation results with our model in the next subsection, it is appropriate to ask: what can we say about the convergence etc. of this identification scheme.

We feel there are two aspects to this question. First, we can ask whether the approximations made in deriving the algorithm in Section II-C are valid and whether we can expect the algorithm to do a proper gradient following, etc. At present, all we can say is that the approximation obtained by unfolding the network in time by only one step seems to be satisfactory (see the results in Section III-B). More analysis is needed to know for what class of plants this approximation may not be satisfactory.

The second and more important aspect of the question of convergence concerns the basic technique of identification using gradient methods and Neural Network models. All such methods calculate the gradient of the instantaneous error $e(k)$ (i.e., error between plant and network outputs at time k) and use that to update the weights. But what is the justification for following the gradient of instantaneous error? Or, more importantly, in what way does $e(k)$ reflect the error in the identification process? We would like to stress that this question is different from the one of batch *vs* incremental learning in pattern recognition problems because the plant that generates the teaching signals is dynamic; hence, the error at a given time is a function of the history and may not really say anything about the performance of the model on the input-output pair at that instant.

To justify the identification method, we need to show that following the gradient of instantaneous error will result in the algorithm minimizing some meaningful measure of error. In this section, we show that an algorithm that follows the gradient of instantaneous error will indeed result in minimizing the expected value of the error between plant and network if the training input given to the plant is independent and identically distributed (*iid*) sequence, and the unknown plant is bounded-input bounded-output (BIBO) stable and controllable.

This, however, does not completely prove the correctness of the identification method. Suppose we use an *iid* input sequence drawn uniformly from $[-1, 1]$ to train the network. Then, we can prove, as stated above, that the algorithm minimizes expected value of error between plant and network for this random input. But what can we say about the network representing the plant accurately for other signals? This is the problem of whether we are giving "sufficiently rich" input during training so that the learning procedure will converge to a network that is closer to plant than any other network. We will not be addressing this question here.

Let $Z(k)$ be the vector denoting the set of all parameters in the network at instant k, which are updated by the algorithm. This consists of all the weights and the memory coefficients. The learning algorithm described in Section II–C can be written as below:

$$Z(k+1) = Z(k) + \eta\, G(Z(k), \psi(k)) \tag{20}$$

Here, η is the step size and

$$\begin{aligned}\psi(k) = (&u(k); y_p(k-1); s_j^\ell(k-1), s_j^\ell(k), v_j^\ell(k-1),\\ &v_j^\ell(k), 1 \le j \le N_\ell, 1 \le \ell < L;\\ &s_n^L(k-1), s_n^L(k), v_{ni}^L(k-1), v_{ni}^L(k),\\ &1 \le i \le M_n, 1 \le n \le N_L).\end{aligned} \tag{21}$$

In (20), the function $G(.,.)$ represents the gradient of the instantaneous error. We will write it symbolically as

$$G(Z(k), \psi(k)) = \frac{\partial e(k)}{\partial Z} \tag{22}$$

It should be noted that this derivative of instantaneous error is a function of all the weights and memory coefficients and the various signals in the network and is evaluated at $(Z(k), \psi(k))$. Consider this function G with $(Z(k)$ held constant at, say, z. If the input signal $u(k)$ is a stationary signal and if the other signals (output of the plant and the outputs of various network and memory neurons) are also stationary, then it makes sense to ask what is the expected value of this instantaneous error where expectation is with respect to the stationary distributions of all the signals involved. This is a good measure of the correctness of the weight values represented by z and hence we want the algorithm to follow the gradient of such an averaged error (note that the averaged error is now a function of the network weights alone). This is what we are going to prove now. The method we use is from [17, Section 5.4] and the reader should consult [17] for details regarding the constructions given below.

The equations (20) and (22) do not fully represent the learning algorithm given in Section II-C because we need to project all the memory coefficients into an interval, say, [0.01, 0.99] to ensure stability of the network. To include such constraints, let $g_i(Z)$, $1 \le i \le m$, be functions such that

$$L = \{Z : g_i(Z) \le 0, \quad 1 \le i \le m\}$$

represents the set of feasible values for the weights. We will call g_i constraints. We will say that the constraint g_i is active at $Z \in L$ if $g_i(z) = 0$. Now we can write our learning algorithm as

$$Z(k+1) = \pi_L(Z(k) + \eta G(Z(k), \psi(k))) \tag{23}$$

where $\pi_L(y)$ is any closest point in L to y. Define the cone $C(z)$ by

$$C(z) = \left\{ y : y = \sum_{i \in A(z)} \lambda_i \partial g_i(z)/\partial z, \lambda_i \geq 0 \right\} \tag{24}$$

where $A(z)$ denotes the set of active constraints at any z that is on the boundary of the constraint set. For points z in the interior of $L, C(z)$ is empty (by definition).

Define the continuous time interpolation of the $\{Z(k), k \geq 0\}$ process as below:

$$Z^n(t) = Z(k) \text{ for } t \in [\eta k, \eta(k+1))$$

Assume that the unknown plant is BIBO stable and controllable. Also assume that the input sequence $u(k)$ is *iid* with some distribution having a compact support. (The input process is hence, trivially, a stationary process). We derive an ordinary differential equation (ODE) associated with the learning algorithm using the following result based on weak convergence analysis [17].

Theorem 1 Under the given assumptions on the input $u(k)$ and the unknown plant, as $\eta \to 0$, the interpolated process $Z^\eta(t)$ converges weakly to the unique solution $z(t)$ of the projected ODE,

$$\dot{z} = \bar{G}(z) + g(z) \tag{25}$$

where $g(z)$ takes values in a cone $-C(z)$ and $\bar{G}(z)$ is the expectation of $G(z, \psi)$ with respect to the invariant distribution of the process $\{\psi(k), k \geq 0\}$. (Here $\dot{z}$ denotes the derivative w.r.t. t).

Proof: The proof of this theorem, which consists of verifying a set of conditions needed to apply a theorem from [17], is given in appendix.

To understand the theorem, consider the ODE given by (25) at a point in the interior of L. That is simply a gradient following ODE where the gradient is that of the "averaged" value of the instantaneous error as explained above (assuming of course that the interchange of derivative and integration are permitted). Thus, if the learning parameter η is small then the algorithm has the same asymptotic behavior as a gradient following algorithm that uses the proper averaged error rather than instantaneous error. For such an averaging effect to take place, it is easy to see that the input sequence should be stationary. In addition, we needed BIBO stability and controllability of the plant to ensure that all the other signals in the network are stationary and thus to ensure the existence of invariant measure as needed for the above theorem. At the boundary points of $L, g(z)$ plays a role in ODE (25) as needed to keep the memory coefficients bounded. One way to understand this is to realize that we want to solve a constrained optimization problem of minimizing the averaged error while keeping memory coefficients between 0 and 1. Hence, we want to reach a *Kuhn-Tucker* point which is not necessarily a zero of the gradient of error. The strategy of this analysis is very similar to that of the standard back propagation algorithm [18]. This technique of analysis making use of weak convergence results is a powerful tool to understand the asymptotic behavior of adaptive algorithms. We refer the reader to [19] for a better appreciation of the technique.

TABLE I
IDENTIFICATION WITH MNN

Example	Size*	Error†	Figure Number
Example 3.1	6:1	0.0752	3
Example 3.1	3:0	0.0668	3
Example 3.2	6:1	0.0641	4
Example 3.2	3:0	0.0345	None
Example 3.3	6:1	0.0186	5(a) (y_{p1})
Example 3.3	6:1	0.0327	5(b) (y_{p2})

**Size* is to the size of the network.
†*Error* is the mean square error between the plant and reference outputs over 1000 time-steps of the test signal. The actual outputs are shown in corresponding figures.

As discussed earlier, this only justifies the procedure of following the gradient of the instantaneous error. For proving full correctness of the identification method we need to properly characterize what constitutes "rich input" for nonlinear systems so that the identified model is valid for all signals.

B. Simulations

In this section we will present a few examples of nonlinear plants identified by Memory Neuron Networks. We use a series parallel model for identification. The structure is shown in Fig. 2 for SISO plants. (We use similar structure for MIMO plants.)

We use networks with only one hidden layer. Hence we use the notation $m : n$ to denote a network that has m hidden network neurons and has n memory neurons per node in the output layer. The number of input and output nodes are determined by the nature of the plant. For SISO plants we will have two inputs ($u(k)$ and $y_p(k-1)$) and one output ($\hat{y}_p(k)$). The number of inputs to our identification model does not depend on the order of the plant.

Example Problems: The examples discussed below are from [3], [4]. The main reasons for this are that they provide fairly complex nonlinear systems and that all of them are known to be stable in the BIBO sense. We also feel that for better understanding of neural networks for control, the various techniques should be tried on the same set of plants. We have tested our identification method on many plants (see [20]), including all the examples given in [3], [4]. We have got good results and the identification algorithm is found to be quite robust. Here we present results for only three examples. The first plant is with geometric nonlinearity. The next example shows the ability of MNN to identify a complex nonlinear system made up of a linear dynamical system and nonlinear memoryless transformation combined [4]. However, unlike the method in [4], our identification algorithm has no knowledge of the structure of the plant or of any of its subsystems. We use the same network and training sequence to identify these plants also. These two are SISO plants. The last example is an MIMO system. Table I summarizes the results of the three examples presented here. (*Note:* We refer the readers to the technical report [20] for more details on the simulations performed).

Network Parameters: As discussed earlier, the memory neurons in the output layer do not play any significant role here. We have used either one or zero memory neurons. We present results for one large and one small network for all the plants below, the two networks being 6:1 and 3:0. We keep the same learning rate for all problems with $\eta = 0.2$ and $\eta' = 0.1$. Also, we have used the same activation functions in all problems—g_1 for hidden nodes and g_2 for output nodes with $c1 = c2 = 1$ and $k_1 = k_2 = 1$ (cf. (3) in Section II–B). Hence, we introduce an attenuation constant in the plant's output so that the teaching signal for the network is always in $[-1, 1]$. This does not affect the identification process but makes for convenience and uniformity.

As is easy to see, the intention here is to explore the generality of the network structure. Thus, any problem-specific tuning would be in terms of the number of learning iterations.

Training the network: We use 62 000 or 77 000 time steps for training the network with the longer training sequence for more complex plants. We train the network for 2000 iterations on zero input; then for two thirds of the remaining training time, the input is *iid* sequence uniform over $[-2, 2]$ and for the rest of the training time, the input is a single sinusoid given by $\sin(\pi k/45)$. For all the plants we have considered, this training appears to be sufficient.

After the training, we compare the output of the network with that of the plant on a test signal for 1000 time steps. Our test signal consists of mixtures of sinusoids and constant inputs. (See (27) below).

Example 3.1: This example clearly indicates the ability of the memory neuron network to learn a plant of unknown order. Here the current output of the plant depends on three previous outputs and two previous inputs as given below.

$$y_p(k+1) = f(y_p(k), y_p(k-1), y_p(k-2), u(k), u(k-1)) \quad (26)$$

where

$$f(x_1, x_2, x_3, x_4, x_5) = \frac{x_1 x_2 x_3 x_5 (x_3 - 1) + x_4}{1 + x_3^2 + x_2^2}$$

Though the function f has five arguments, we feed to the network, only $u(k+1)$ and $y_p(k)$ for it to output $\hat{y}_p(k+1)$. Through the process of learning, the network has evolved the right values for the memory coefficient to be able to reproduce the behavior of (26). For the test phase, we used the following input

$$\begin{aligned} u(k) &= \sin(\pi k/25), k < 250 \\ &= 1.0,\ 250 \le k < 500 \\ &= -1.0, 500 \le k < 750 \\ &= 0.3\sin(\pi k/25) + 0.1\sin(\pi k/32) \\ &\quad + 0.6\sin(\pi k/10),\ 750 \le k < 1000 \end{aligned} \quad (27)$$

Fig. 3 shows the output of the plant and the two network models (6:1 and 3:0) for the test input given by (27).

It should be noted here that if we use a feed-forward network for learning this plant then we need to have five input nodes to feed the appropriate past values of y_p and u.

Example 3.2: The plant is given by

$$y_p(k+1) = f(x_p(k)) \quad (28)$$

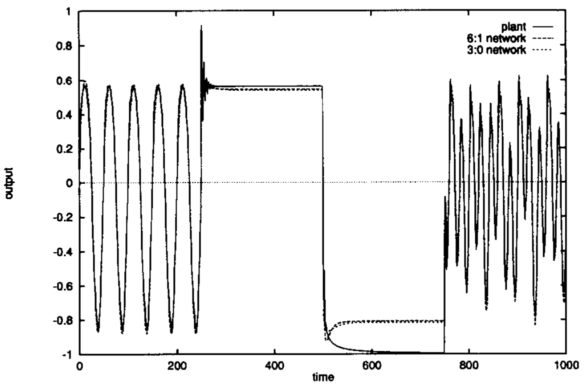


Fig. 3. Output of the plant and model network for Example 3.1. Output of a 6:1 network and a 3:0 network are shown in the figure.

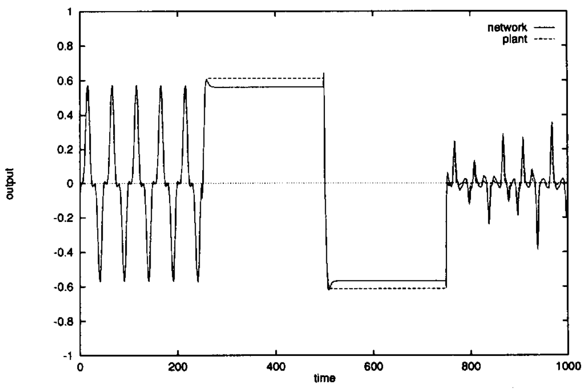


Fig. 4. Output of the plant and model network (6:1) for Example 3.2.

where

$$\begin{aligned} f(v) &= \left(\frac{v}{3.5}\right)^3 \\ x_p(k) &= W(z)h(u(k)) \\ W(z) &= \frac{z + 0.3}{z^2 - 0.8z + 0.15} \\ h(v) &= \frac{4v^3}{1 + 4v^2} \end{aligned} \quad (29)$$

Here the plant is specified through the combinations of linear filters and nonlinear memoryless transformations. In the above equations, $W(z)$ is the transfer function of a linear discrete time system. There is a little abuse of notation in the specification of $x_p(k)$ above and it is to be understood in the usual sense of interpreting z as a time shift operator. Here $z^{-1}h(u(k))$ will represent $h(u(k-1))$. Fig. 4 shows the results using the test sequence (27) for the 6:1 network.

Example 3.3 Our final example is an MIMO plant with two inputs and two outputs. Thus we will use a network with four network neurons in the input layer and two network neurons in the output layer. For this example our network structure is 6:1. The plant is specified by

$$y_{p1}(k+1) = 0.5\left[\frac{y_{p1}(k)}{1 + y_{p2}^2(k)} + u_1(k)\right]$$

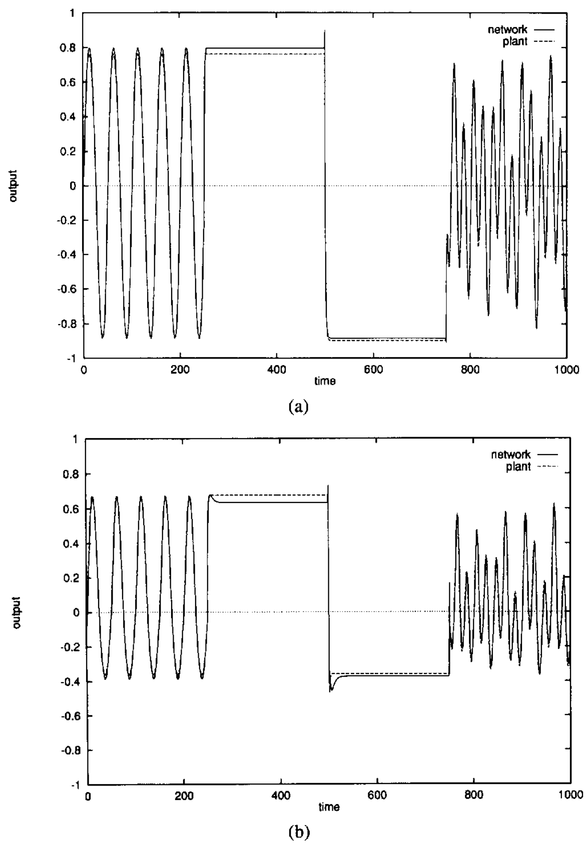


Fig. 5. Output of the MIMO plant and model network for Example 3.3; (a) shows the first output of a 6:1 network and (b) shows the second output of the same network.

$$y_{p2}(k+1) = 0.5\left[\frac{y_{p1}(k)y_{p2}(k)}{1+y_{p2}^2(k)} + u_2(k)\right] \quad (30)$$

Fig. 5(a) and Fig. 5(b) show the output of the plant and network for the test signal given by (27). As is easy to see, the identification is good for the MIMO plant also. Table I gives the actual error between the plant and the network outputs for all these examples.

IV. Adaptive Control of Dynamical Systems Using MNN

In contrast to linear systems, for which there now exists considerable theory regarding adaptive control [21], very little is known concerning adaptive control of plants governed by nonlinear equations. Thus, like in many other areas, applications of neural networks for control is largely driven by empirical studies [2]. In this section, we discuss model reference adaptive control using memory neuron networks. We assume that the plant is stable in the BIBO sense and that there is sufficient knowledge about the plant to specify the goal of control in terms of a reference model.

Given a plant, a reference model and a reference input, the problem is to determine the input to the plant (which will be the output of a neural network controller) so that the output of the plant follows that of the reference model.

In *indirect control,* one identifies the plant using some identification model and then uses the parameters of the learned model to derive the controller network. The method of indirect control relies on the ability to derive the control law given the identified model, for a class of systems (often using the so called certainty equivalence control). The main difficulty with using neural networks for indirect control is the fact that these are nonparametric identifiers [12] and hence there is no simple relationship between the learned weights of the network and the parameters of the plant (even if we had known the form of the nonlinear transformation of the plant. This method is discussed in detail by Narendra and Parthasarathy [3], [4] and MNN can be used for indirect control in a similar way (See example 4.2 in [20]).

In *direct adaptive control,* the parameters of the controller are directly adjusted based on the error between output of the plant and that of the reference model. The main difficulty here is that of credit assignment in training the controller network. We cannot supply a proper error signal to the controller net because the unknown plant lies between the controller and the available error signal.

Various strategies have been tried to overcome this problem. The simplest solution will be if we have a "knowledgeable controller" (e.g., a human or other costly control device) which already knows how to control the plant. However, this is not often practical.

There are at least two general purpose techniques to tackle the problem of training the controller net [2], [13]. Both of them make use of an identification model. However, instead of using the parameters in the identified model to directly derive the controller net, these techniques obtain some signal using the identification model which is then used to train the controller net. We use one of the methods here. Fig. 6 shows the block diagram of the method we discuss here, which is called *differentiating the model* or *forward and inverse modeling* [2], [13]. Here we keep a forward model of the plant (network $N1$ in Fig. 6) which is used as a channel to black-propagate the error at the plant output up to the plant input and this back-propagate the error at the plant output up to the plant input and this back-propagated error is used for training the controller net ($N2$ in Fig. 6). For the technique to work properly, the forward model of the plant should be accurate at all times. So, at each time step we will update the weights of the plant model based on the error (e_I in Fig. 6) between the output of the plant and that of the network $N1$. Then we calculate the error (e_c in Fig. 6) between the output of the plant and that of the reference model, which is then back-propagated through $N1$ to supply the error signal for the controller network.

A. Training the Controller Network

As explained above, we use the method of differentiating the model which is a general purpose control scheme for ANN based methods. The overall structure of the controller is as shown in Fig. 6. However there is one serious difficulty in

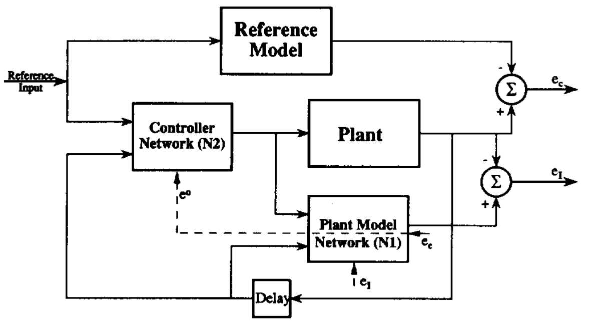


Fig. 6. Model reference adaptive control with neural networks: *Differentiating the plant model.* The error e_I is used for online adaptation of plant model $N1$ and the error e^0 is used for online adaptation of the controller $N2$. e^0 is generated by back-propagating the error e_c through $N1$, up to its input.

back-propagating the error e_c through $N1$ for training the controller network when the plant involves significant delay. Since this has not been pointed out in literature, we explain the problem and suggest a solution.

Consider Fig. 7, which shows an adaptive controller for an SISO plant using feed-forward nets. The figure shows explicitly the delayed inputs that are fed to the network. Let us say we know that the plant output can depend on $u(k-1)$, and $u(k-2)$, and hence these are fed to the net $N1$ by delays from the single output of $N2$. (We will forget about other inputs, such as past outputs of the plant, to $N1$ for now because they are not important to this discussion). For training $N2$, we need to fine the error at the output node of $N2$ which is directly connected to input node 1 of $N1$. So, it seems all we need do is to back-propagate the error, e_c, through $N1$ to reach node 1 in its input layer. But suppose that the plant output depends only on $u(k-2)$. Then $N1$, which has three inputs, would have learned a function that is independent of input 1. Hence if the identification was good, then the back-propagated error will always be zero! Thus, in general if the plant output depends on many previous inputs (or if the plant has some unknown pure delay) then there will be a problem regarding this back propagation of error through $N1$ to train $N2$. We think the correct procedure would be to use the idea of dynamic back propagation [4] to propagate the error through the delay line (which is just a linear filter with transfer function z^{-1}). In practice this would mean we have to back-propagate the error e_c up to all input nodes of $N1$, then add appropriately delayed versions of these errors to get the error at the output node of $N2$. Hence, even if we are using feed-forward nets with incremental learning algorithms, we need to store past values of errors etc. at nodes to implement this back propagation

In the case of Memory neuron networks, since there are no external delay lines this problem will not occur. However if we use the approximation of unfolding in time by exactly one step, as discussed in Section II–C, then we will get into similar problems. Consider the network shown in Fig. 6 with both $N1$ and $N2$ being Memory Neuron Networks of the general structure shown in Fig. 1. If we use the algorithm as given in Section II–C when we back-propagate error e_c to calculate the controller error, we would be back-propagating only through the weights w_{ij}'s and not through the memory

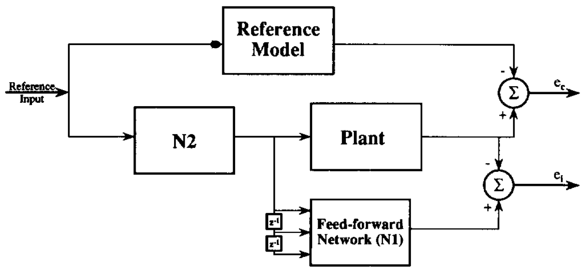


Fig. 7. Problems with training the controller net when the plant involves delay (see text).

neurons (see Section II-C). Hence, once again we would get wrong value of error if the plant output depended only on some past values of input. In case of MNN, the problem is only due to the approximation used in Section II-C in calculating the gradient. Here we can take care of it by back-propagating error through the memory neurons also. That is, when the plant model network is used as a channel for back propagation, we unfold the recurrent network by more than one time step. This results in a better approximation of the gradient than the one given by (9).

Let e^L_1 denote the error at the single output node of network $N1$ (the equations given below are extended in an obvious manner of multiple outputs). Let e^l_j denote the error at the jth network neuron at level ℓ. Let e^o_j be the error at the jth input node of $N1$ and hence for $j = 1$ (say) it will be the e^o that is needed for updating $N2$. The errors are calculated as below. (Recall from Section II-C that x^ℓ_j is the net input to neuron j at level l and g_1', g_2' are the derivatives of the activation functions of the hidden and output nodes).

$$e^L_1(k) = g_2'(x^L_1(k))(y_p(k) - y_m(k))$$
$$e^\ell_j(k) = g_1'(x^\ell_j(k)) \sum_{p=1}^{N_{\ell+1}} e^{\ell+1}_p [w^\ell_{jp} + f^\ell_{jp}\alpha^\ell_j], \quad 1 \le \ell < L,$$
$$e^0_j(k) = \sum_{p=1}^{N_1} e^1_p(k)[w^0_{jp} + f^0_{jp}\alpha^0_j] \qquad (31)$$

It is easy to see that (31) is still only an approximation to the actual gradient. Otherwise we have to include terms $f^\ell_{jp}(1-\alpha^\ell_j)\alpha^\ell_j$ etc. in the error back propagation equations. However, the above approximation seems to work well in practice. As a matter of fact, if we use (9) for calculating $e^\ell_j, l > 0$ and use (31) only for calculating e^0, we still get good performance with the adaptive controller. In (31) we have deliberately left out dependence on k of the weights because for calculating this error we are assuming that the plant model is fixed. In practice, though the weights in $N1$ do get updated, the amount of change during the online adaptation would be very small.

B. Simulations

In this section we describe the results obtained using MNN as controllers for some nonlinear plants. The general structure is shown in Fig. 8. The figure is for SISO plants and hence both networks have two inputs and one output. At time step

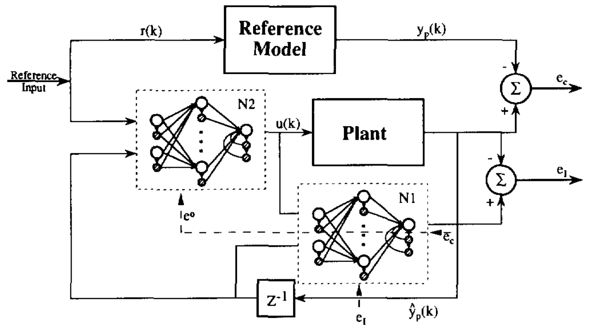


Fig. 8. Model reference adaptive control with Memory Neuron Networks. Adaptation is same as in Fig. 6. Here only *one* previous output of the plant is fed into networks $N1$ and $N2$.

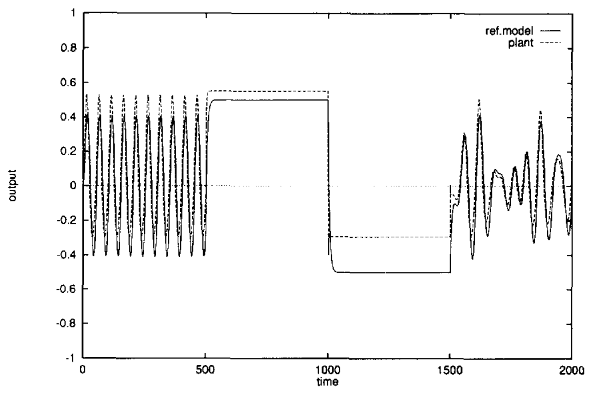
$k, r(k)$ is the reference input, $u(k)$ is the input to the plant (which is the output of the controller net), $y_p(k)$ and $\hat{y}_p(k)$ are the output of the plant and the model net $N1$, and $y_m(k)$ is the output of the reference model.

Training of the controller proceeds as follows. We start with *off-line identification phase* where we train the network $N1$, using the methods described in Section III-A. Then we have an *off-line training* phase for the controller where cascade of $N2$ and $N1$ are trained to mimic the reference model. Here we update only the weights of $N2$ using $N1$ as a channel of back propagation It should be noted that the plant is not involved in this training. For this off-line training, $r(k)$ is taken to be *iid* uniformly distributed over $[-1, 1]$ and this training is continued for about 10 000 to 20 000 time-steps. Then we will connect the plant (as in Fig. 8) and train the controller online. We get fairly good results even if we did not use the off-line training, the controller is able to input fairly accurate control inputs right from the beginning when connected with the plant. Since, off-line training does not involve the plant, this seems to be a better strategy. During online adaptation of the controller, (referring to Fig. 8), we use e_I to update $N1$, back-propagate e_c through $N1$ to calculate error at output of $N2$, denoted by e^0, and use this error to update $N2$. We use the algorithm as in Section II-C to update $N1$ using e_c and $N2$ using e^0. However, as explained in Section IV-A, for calculating the error, e^0, we use (31).

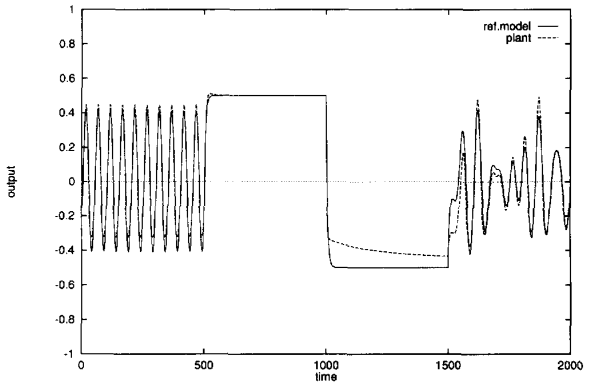
We now describe several examples, where the scheme as discussed above is utilized for adaptively controlling some nonlinear plants. As in Section III-B, most of these examples are also from [3], [4]. In these examples, training of network $N1$ proceeds in the same fashion as in Section III-B. For training network $N2$, we use a smaller learning rate. Typically the parameters η, η' for $N2$ are between 0.25 to 0.5 times the values of corresponding parameters for $N1$ (which are 0.2 and 0.1 respectively). It is observed that, unlike the case of identification described in Section III-B, performance of the controller network is sensitive to these parameter values.

Example 4.1: The plant is given by

$$y_p(k+1) = 0.35\left[\frac{y_p(k)y_p(k-1)[y_p(k)+2.5]}{1+y_p^2(k)+y_p^2(k-1)} + u(k)\right] \tag{32}$$



(a)



(b)

Fig. 9. Outputs of reference model and the plant in example 4.1. (a) shows the output of the uncontrolled plant and (b) shows the output of the plant with the controller.

The reference model is a second order linear system given by

$$y_m(k+1) = 0.6y_m(k) + 0.2y_m(k-1) + 0.1r(k)$$

As in Section III-B, we have used some constant gains to keep the output of the plant and reference model in $[-1, 1]$ for all inputs that we encounter. For testing the adaptive controller, we use the following reference input.

$$\begin{aligned} r(k) &= \sin(\pi k/25),\ k < 500 \\ &= 1.0,\ 500 \le k < 1000 \\ &= -1.0,\ 1000 \le k < 1500 \\ &= 0.3\sin(\pi k/25) + 0.4\sin(\pi k/32) \\ &\quad + 0.3\sin(\pi k/40),\ k \ge 1500 \end{aligned} \tag{33}$$

Fig. 9(a) shows output of the reference model and the uncontrolled plant (where $u(k) = r(k)$). We use the structure in Fig. 8 for control. $N1$ is a 6:1 network, off-line trained as in Section III-A. $N2$ is also a 6:1 network. The activation function for output node of $N2$ has range $[-2, 2]$ (i.e., $c2 = 2$ in (3)). This provides sufficient range at the input of the plant for it to follow the reference model. As explained earlier, $N2$ is off-line trained without the plant for 20 000 time steps using *iid* input uniformly over $[-1, 1]$. We then connect the plant and keep $r(k) = \sin(\pi k/45)$ for another 20 000 time steps.

We then test the online control using $r(k)$ given by (33). The output of the controlled plant and that of the reference model are shown in Fig. 9(b). Comparing this with Fig. 9(a), it is easy to see the control is quite effective.

We next investigate the ability of the controller to take care of changes in the plant online We train our controller as earlier and then start $r(k)$ given by (33) at time 0. Then at time 150, we change the constant 2.5 in the plant equation to zero. At time 800, we change the plant equation to

$$y_p(k+1) = 0.35\left[\frac{y_p^2(k)[y_p(k-1)+2.5]}{1+y_p^2(k-1)} + u(k)\right] \quad (34)$$

Fig. 10(a) shows the output of reference model and the uncontrolled plant with the above online changes. Comparing this with Fig. 9(a) gives an idea of the changes that need to be compensated online by our controller. Fig. 10(b) shows the output of the controlled plant and that of the reference model. It is easy to see that, though the control goes wrong when the plant changes, the controller is able to adapt itself online to the changes. For this part, the learning rate parameter of the network $N2$ is doubled after 2000 steps of online adaptation (which is before the test signal is input to the system). Fig. 10(c) shows the performance when there is no online adaptation of the controller, that is the weights in the network $N2$ are fixed after the initial training phase.

Example 4.2 The plant is given by

$$y_p(k+1) = 0.5\left[\frac{y_p(k)}{1+y_p^2(k)} + (1+u(k))u(k)(1-u(k))\right] \quad (35)$$

Here the output is a nonlinear function of the input that is not invertible. The reference model is a first order linear system given by

$$y_m(k+1) = 0.6y_m(k) + 0.15r(k)$$

We use the same procedure as in Example 4.1 to train the controller. Fig. 11 shows its performance when tested with reference input given by (33). It may be noted here that the output of the plant without the controller would be zero when the reference input is $+1$ or -1.

V. Discussion

In this paper we have suggested Memory Neuron Networks as general models for identification and control of dynamical systems. The main attraction of these networks is that they have internal memory and hence are themselves dynamical systems. Most of the neural network models used for identification and control are feed-forward networks. Since feed-forward nets can only represent memoryless transformations, one needs to explicitly feed all the past inputs and outputs of the plant to the network model through explicit delays. This is not wholly satisfactory on two grounds.

First, we need to know the exact order of the system to be able to feed the right set of inputs to the network model. This limits the utility of these models and also may make them inefficient due to large number of inputs needed. The second problem is that using static networks to model dynamical systems is inherently unsatisfactory. (See also the discussion in [9]).

In this sense, Memory Neuron Networks offer truly dynamical models. The memory neurons are sensitive to history and the memory coefficient in the network are the parameters that control what past values can affect the current output. Moreover these memory coefficients are modified "online"

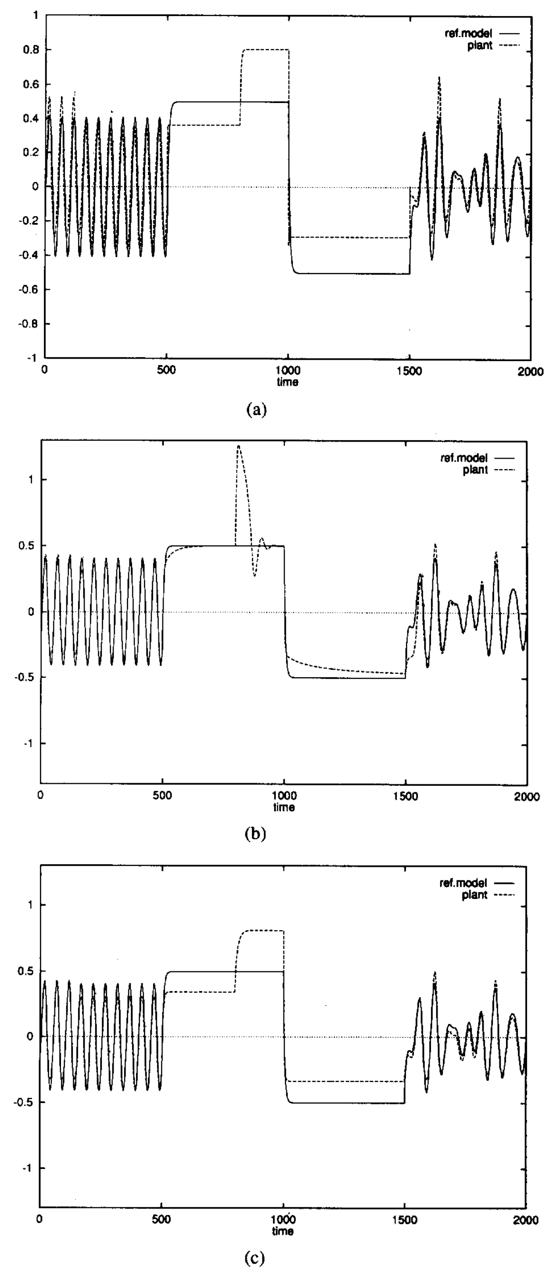


Fig. 10. Output of reference model and plant with online changes for Example 4.1. (see text). (a) shows the output of the uncontrolled plant, (b) shows the output of the plant with a MNN controller that is adapted online, and (c) shows the plant output when the controller is not adapted online.

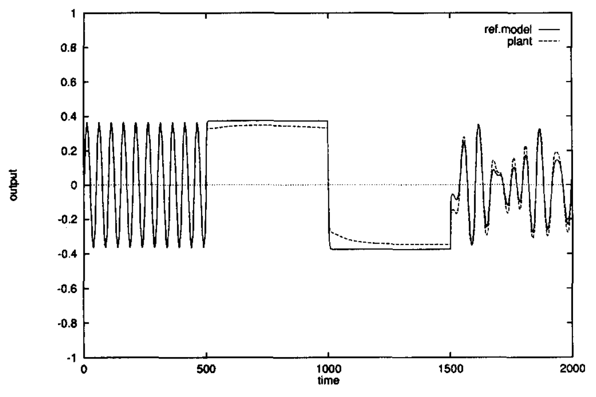


Fig. 11. Outputs of reference model and the controlled plant in Example 4.2, for the reference input given by (33).

during the learning process [6]. The model is much simpler compared to a general recurrent network which allows for arbitrary feedback connections [16]. Here the network has a "near-to-feed-forward" structure which is useful both for the heuristic design of the network architecture and for having an incremental learning algorithm that is fairly robust. Following Back and Tsoi [22], we can consider Memory Neuron Networks to be a "locally" recurrent and "globally" feed-forward architecture that can be considered as intermediate between feed-forward and general recurrent networks.

As discussed in Section I, many initial efforts at controlling dynamical systems using Neural Networks made use of feed-forward networks with tapped delay lines [9] on the input to capture dynamics. One way of improving this delay line network model is to keep a fixed number of delays, but adaptively modify the amount of delay. Day and Davenport [23] recently suggested a continuous time temporal back propagation algorithm with adaptive time delays. This model requires *a priori* knowledge of the plant, to decide on the number of delay links required in the network. Leighton and Conrath [24] suggested the Auto-Regressive model neuron (AR neuron) which consists of a nonlinear element in cascade with a linear filter. This filter could be of some nth order and the feedback connections through delays helps in storing the necessary past information. Input to the nonlinear element is just the current input. Back and Tsoi [22] have suggested an architecture in which inputs to the nonlinear element are obtained after passing through a FIR or IIR filter (referred as FIR/IIR synapse). Memory Neuron Networks are similar to the above architectures except that 1) they use simple first order filters (memory neurons) in cascade and 2) direct links are provided from the output of the filters as well as the nonlinear elements to the next layer neurons.

There are no analytical results, at present, regarding the representational capability of any of these dynamical models. Filter parameters are adjusted online in all of the above models. As pointed out in [24], ensuring the stability of a general nth order filter requires extra non-trivial computations (e.g., Ruth-Hurwitz criterion to test stability during each update), while it is simple for a first order filter as used in MNN. We refer to the recent review by Nerrand *et al.* [25] for a unified view of such network architectures (both feed-forward and recurrent models).

Gradient based learning algorithms for recurrent networks are not direct as in the case of feed-forward networks since the output is affected by the current as well as past inputs. To account for the effects of the past inputs, the weights can be held constant over a fixed time interval in order to minimize some integral squared error criterion [16]. Required gradient information is accumulated over this time interval and the weights are updated at the end of the time interval. Many "episodes" of such updates over fixed intervals are used for adaptively learning the weights and the length of this interval depends on the application [16], [24].

For real-time application it will be convenient to minimize only the instantaneous error rather than the integral error. For sufficiently small step size, if the weight changes are small, then the instantaneous updates can be considered as a close approximation to the (above) method of updates over fixed intervals. Such algorithms recursively use the gradient values calculated at previous time steps to compute the current weight change [16]. The past gradient values used in these procedures are computed using the weight values in the past. These recursive first-order gradient algorithms can thus be considered equivalent to the algorithm which performs back propagation through n-time steps, added with the corresponding momentum terms from the past n-time steps. For the Memory Neuron Networks which uses first order filters, Poddar and Unnikrishnan [6], [11] have suggested a similar gradient computation procedure.

For the purpose of this work, we have used only back-propagating through one time step for updating the parameters of MNN and recursive computation is not used. Though the procedure is simplified, in all our simulations it performs very well on various plants considered for identification and control [20]. Further work is needed to decide what level of approximation is appropriate for the problem of adaptive control of dynamical systems.

Unlike in pattern classification problems where one can cycle through the training data repeatedly, for identification and adaptive control we need incremental learning algorithms. Thus at each time step we have to use the instantaneous error at that time for updating the weights. Since the teaching signal itself is the output of an unknown dynamical system, an important question that needs to be answered is: what does following the gradient of the instantaneous error lead to. We have shown that if the unknown plant is BIBO stable and controllable, and the random test input is stationary with a distribution that has compact support, then asymptotically we will be minimizing a meaningful error measure.

However, this analysis of the identification algorithm is still incomplete. We have shown that the algorithm will minimize the expected value of the error (between the outputs of plant and model network) where the expected value is with respect to the stationary distribution of the test input. This does not answer the question of whether the test signal chosen (in our case *iid* with uniform distribution) is good enough for the model network to mimic the plant for all signals. Further theoretical analysis is needed to prove the correctness of identification procedures using neural networks.

For the control scheme we have used a network model for the forward dynamics of the plant as a channel for back-propagating the error in the plant output to be able to update the controller net. This technique has been used by many other researchers [7], [8], [26]. However, there is one difficulty in this back-propagation if the plant involves some delay. We have discussed this problem and have suggested a solution both for feed-forward controllers and for Memory Neuron Networks. We have not presented any theoretical analysis of the control algorithm. At present there are no known theoretical results regarding convergence of neural network algorithms for control of nonlinear systems. For the control technique discussed in Section IV (see Fig. 8), the presence of two networks and separate updating make the analysis difficult. To get an appreciation of the difficulties involved, consider the simple case of a constant reference input (i.e., a regulation problem). Assuming correct identification, (i.e., the model $N1$ is correct), the controller will learn to minimize error e_c. However, if the plant characteristic change during the operation, we need to examine the combined dynamics of $N1$ and $N2$ to be sure of the correctness of control. Once again if we assume that changes in $N1$ occur much faster then $N2$, then at any time the identification is complete before the controller is significantly changed. Hence we need also to be aware of the problem of stability. Assuming BIBO stability of the plant, we can keep the plant output always bounded by using a sufficiently small bounded output of the controller through a proper sigmoid function. But this assumes knowledge of the plant and it would also mean we cannot control the plant for all possible reference inputs. Thus, even in this simple case, much hand-waving is needed to justify the neural network algorithms.

More work needs to be done, especially in analysis of neural network models, before such networks are routinely used for controlling nonlinear systems. We feel that endowing feed-forward nets with dynamics is a small first step in coming out with a tractable class of dynamical models with neural networks. Hopefully the current interest in the field will lead to work in that direction.

VI. Conclusion

In this paper we have discussed identification and control of nonlinear dynamical systems using Memory Neuron Networks. We are able to identify a variety of complex nonlinear systems using the *same* network structure. As can be seen from the results of Section III-B, this method is quite robust. We have given some theoretical justification of our identification procedure.

We have also presented simulations using these networks for model reference adaptive control. We have used the technique of differentiating the plant model for training the controller. This method as currently used [2], [13] has a serious shortcoming. We have pointed this out and have suggested a solution. We have shown the effectiveness of these controllers for both tracking the reference input and for adapting to changes in the plant.

Appendix

A. Proof of Theorem.

The proof essentially verifies the conditions for applying a result due to Kushner (Theorem 5.5, [17]). We observe the following with respect to the given algorithm and the neural network model.

- The process $\{Z(k), \psi(k-1), k > 0\}$ is a Markov process. This follows from the algorithm and the definition of the vector ψ. Also note that the process $\{u(k)\}$ is chosen to be an *iid* process.
- The function $G(.,.)$ is continuous in both the arguments as can be seen from the algorithm. Hence the function is bounded over any compact set.
- The function $G(.,.)$ is independent of the step parameter $\eta > 0$ and the one step transition probability function $P(\psi, 1, B|z)$ for some borel set B on the appropriate space is independent of the step size η, for a fixed value of the network parameters z.
- The input $\{u(k)\}$ is an *iid* process and hence it is trivially stationary. Since the plant is assumed to be stable and controllable the output of the plant $\{y_p(k)\}$ is stationary [27]. Using the above observations, since the input to the network is stationary, for a fixed set of parameters, z, of the network the process $\{\psi(k)\}$ is also stationary. Hence there is unique invariant distribution M^z for the process $\{\psi(k)\}$.
- Since the input process is chosen to be an *iid* process it is trivially tight. The other input to the network $y_p(k)$ takes values in a compact set since the plant is assumed to be BIBO stable. All the other signals in the network also take values inside a compact set. In fact, the activation function for the network units is the logistic function $f(x) = 1/(1 + \exp(-x))$, which is bounded, and the memory coefficients (α) of the memory units take a value strictly less than one and greater than zero, which makes the memory units stable. Hence all the signals in the network take values inside a compact set. It follows that the set of all invariant distributions $\{M^z\}$ is tight.

From the above observations it follows, using the result (Theorem 5.5, [17]), that as $\eta \to 0$, the interpolated process $Z^n(t)$ weakly converges to $z(t)$ which satisfies the projected ODE,

$$\dot{z} = \overline{G}(z) + g(z)$$

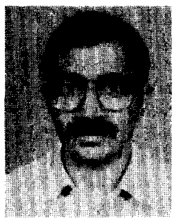
here $g(z)$ takes values in a cone $-C(z)$ for all boundary points z. If z is an interior point then $g(z)$ is zero.

Acknowledgment

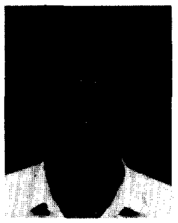
The authors thank Hossein Javaherian for many helpful discussions. The clarity of the paper has been improved by the comments of Hossein Javaherian and James Elshoff. Harmon Nine helped us in generating the figures and Lalitha had done some early simulation of MNN for identification. Most of this work was done while the first author (PSS) was visiting the University of Michigan, Ann Arbor, and General Motors Research Laboratories, Warren, MI.

REFERENCES

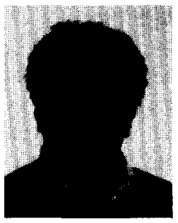
[1] W. T. Miller, R. S. Sutton, and P. J. Werbos, *Neural Networks for Control*. Cambridge, MA: MIT Press, 1990.
[2] A. G. Barto, "Connectionist learning for control: An overview," in *Neural Networks for Control* W. T. Miller, R. S. Sutton, and P. J. Werbos, Eds. MIT Press, Cambridge, 1990.
[3] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.
[4] K. S. Narendra and K. Parthasarathy, "Gradient methods for optimization of dynamical systems containing neural networks," *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 252–262, 1991.
[5] Y. Ichikawa and T. Sawa, "Neural network application for direct feedback controllers," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, pp. 224–231, 1992.
[6] P. Poddar and K. P. Unnikrishnan, "Memory neuron networks: A prolegomenon," *Tech. Rep. GMR-7493*, General Motors Research Laboratories, 1991.
[7] D. A. Hoskins, J. N. Hwang, and J. Vagners, "Iterative inversion of neural networks and its application to adaptive control," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, pp. 292–301, 1992.
[8] Q. H. Wu, B. W. Hogg, and G. W. Irwin, "A neural network regulator for turbogenerators," *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 95–100, 1992.
[9] R. J. Williams, "Adaptive state representation and estimation using recurrent connectionist networks" in *Neural Networks for Control* W. T. Miller, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT Press, 1990.
[10] K. P. Unnikrishnan, J. J. Hopfield, and D. W. Tank, "Connected-digit speaker-dependent speech recognition system using a neural network with time-delayed connections," *IEEE Transactions on Accoustics Speech and Signal Processing*, vol. 39, pp. 698–713, 1991.
[11] P. Poddar and K. P. Unnikrishnan, "Efficient real-time prediction and recognition of temporal patterns," in *IEEE Workshop on Neural Networks for Signal Processing*, Princeton, USA, October 1991.
[12] S. Geman, E. Bienenstock and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural Computation*, vol. 4, pp. 1–58, 1992.
[13] M. Kawato, "Computational schemes and neural network models for formation and control of multijoint arm trajectory," in *Neural Networks for Control* W. T. Miller, R. S. Sutton, and P. J. Werbos eds. Cambridge, MA: MIT Press, 1990.
[14] M. I. Jordan and D. E. Rumelhart, "Forward models: Supervised learning with a distal teacher," *Cognitive Science*, vol. 16, pp. 307–354, 1992.
[15] K. P. Unnikrishnan and K. P. Venugopal, "Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks," in *Neural Computation*, vol. 6, pp. 467–488, 1994.
[16] R. J. Williams and D. Zipser, "A learning algorithm for continually running recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.
[17] H. J. Kushner, *Approximation and Weak Convergence Methods for Random Process*. Cambridge, MA: MIT Press, 1984.
[18] C.-M. Kuan and K. Hornik, "Convergence of learning algorithms with constant learning rates," *IEEE Transactions on Neural Networks*, vol. 2, pp. 484–489, 1991.
[19] A. Beneveniste, M. Metivier, and P. Priouret, *Adaptive Algorithms and Stochastic Approximations*. New York: Springer Verlag, 1987.
[20] P. S. Sastry, G. Santharam, and K. P. Unnikrishnan, "Memory neuron networks for identification and control of dynamical systems," *Tech. Rep. GMR-7916*, General Motors Research Laboratories, 1993.
[21] K. S. Narendra and A. M. Annaswamy, *Stable Adaptive Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[22] A. D. Back and A. C. Tsoi, "FIR and IIR synapses, a new neural network architecture for time series modeling" *Neural Computation*, vol. 3, no. 3, pp. 375–385, 1991.
[23] S. P. Day and M. R. Davenport, "Continuous time temporal back propagation with adaptable time delays," *IEEE Transactions on Neural Networks*, vol. 4, pp. 348–354, 1993.
[24] R. R. Leighton and B. C. Conrath, "The autoregressive backpropagation algorithm," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, pp. 369–377, 1991.
[25] O. Nerrand, P. Roussel-Ragot, L. Personnaz, G. Dreyfus, and S. Macros, "Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms," *Neural Computation*, vol. 5, pp. 165–199, 1993.
[26] D. Nguyen and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," in *Neural Networks for Control* W. T. Miller, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT Press, 1990.
[27] S. P. Meyn and P. E. Caines, "Asymptotic behavior of stochastic systems possessing Markovian realisations," *SIAM Journal of Control and Optimization*, vol. 29, pp. 535–561, May 1991.



P. S. Sastry, (S'82–M'85), received the B.Sc. (Hons.) in Physics from the Indian Institute of Technology, Kharagpur, in 1978, and the B.E. degree in Electrical Communications Engineering and the Ph.D. degree in Electrical Engineering from the Indian Institute of Science, Bangalore, in 1981 and 1985, respectively. Currently, he is an assistant professor in the Department of Electrical Engineering at the Indian Institute of Science, Bangalore. His research interests include learning algorithms, neural networks, pattern recognition, and artificial intelligence.



G. Santharam received the B.Sc. degree in Physics from Madras University in 1984, and the M.E. degree in Electrical Communications Engineering from the Indian Institute of Science, Bangalore, in 1988. Currently, he is working toward the Ph.D. degree in the department of Electrical Engineering at the Indian Institute of Science, Bangalore. His research interests include learning algorithms, neural networks, pattern recognition, and stochastic systems.



K. P. Unnikrishnan, received the B.Sc. degree in Physics from Calicut University, India, in 1979, the M.Sc. degree in Physics from Cochin University, India, in 1981, and the Ph.D. degree in Biophysics from Syracuse University, Syracuse, NY, in 1987.

From 1987 to 1989 he was a post-doctoral Member of Technical Staff at AT&T Bell Laboratories, Murray Hill, NJ. He is currently a Senior Research Scientist in the Computer Science Department at the General Motors Research Laboratories, Warren, MI, and an Adjunct Assistant Professor in Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI. His research interests concern neural computation in sensory systems, neural mechanisms of learning and development, and neural architectures for control.