Neural Computation vol. 5, 99. 165-197 (1993)

NEURAL NETWORKS AND NON-LINEAR ADAPTIVE FILTERING: UNIFYING CONCEPTS AND NEW ALGORITHMS

O. NERRAND, P. ROUSSEL-RAGOT, L. PERSONNAZ, G. DREYFUS
Ecole Supérieure de Physique et de Chimie Industrielles de la Ville de Paris
10, rue Vauquelin
75005 PARIS - FRANCE

S. MARCOS
Laboratoire des Signaux et Systèmes
Ecole Supérieure d'Electricité
Plateau de Moulon
91192 GIF SUR YVETTE - FRANCE

Abstract

The paper proposes a general framework which encompasses the training of neural networks and the adaptation of filters. We show that neural networks can be considered as general non-linear filters which can be trained adaptively, i. e. which can undergo continual training with a possibly infinite number of time-ordered examples. We introduce the canonical form of a neural network. This canonical form permits a unified presentation of network architectures and of gradient-based training algorithms for both feedforward networks (transversal filters) and feedback networks (recursive filters). We show that several algorithms used classically in linear adaptive filtering, and some algorithms suggested by other authors for training neural networks, are special cases in a general classification of training algorithms for feedback networks.

INTRODUCTION

The recent development of neural networks has made comparisons between "neural" approaches and classical ones an absolute necessity, in order to assess unambiguously the potential benefits of using neural nets to perform specific tasks. These comparisons can be performed either on the basis of simulations - which are necessarily limited in scope to the systems which are simulated - or on a conceptual basis - endeavouring to put into perspective the methods and algorithms related to various approaches.

The present paper belongs to the second category. It proposes a general framework which encompasses algorithms used for the training of neural networks and algorithms used for the estimation of the parameters of filters. Specifically, we show that neural networks can be used *adaptively,* i.e. can undergo *continual* training with a possibly *infinite* number of *time-ordered* examples - in contradistinction to the traditional training of neural networks with a *finite number* of examples presented in an *arbitrary order;* therefore, neural networks can be regarded as a class of non-linear adaptive filters, either transversal or recursive, which are quite general because of the ability of feedforward nets to approximate non-linear functions. We further show that algorithms which can be used for the adaptive training of feedback neural networks fall into four broad classes; these classes include, as special instances, the methods which have been proposed in the recent past for training neural networks adaptively, as well as algorithms which have been in current use in linear adaptive filtering. Furthermore, this framework allows us to propose a number of new algorithms which may be used for non-linear adaptive filtering and for non-linear adaptive control.

The first part of the paper is a short presentation of adaptive filters and neural networks. In the second part, we define the architectures of neural networks for non-linear filtering, either transversal or recursive; we introduce the concept of *canonical form* of a network. The third part is devoted to the adaptive training of neural networks; we first consider transversal filters, whose training is relatively straightforward; we subsequently consider the training of feedback networks for non-linear recursive adaptive filtering, which is a much richer problem; we introduce *undirected, semi-directed,* and *directed* algorithms, and put them into the perspective of standard approaches in adaptive filtering (*output error* and *equation error* approaches) and adaptive control (*parallel* and *series-parallel* approaches), as well as of algorithms suggested earlier for the training of neural networks.

1. SCOPES OF ADAPTIVE FILTERS AND OF NEURAL NETWORKS

1.1. ADAPTIVE FILTERS

Adaptive filtering is of central importance in many applications of signal processing, such as the modelling, estimation and detection of signals. Adaptive filters also play a crucial role in system modelling and control. These applications are related to communications, radar, sonar, biomedical electronics, geophysics, etc.

A general discrete-time filter defines a relationship between an input time sequence {u(n), u(n–1), …} and an output time sequence {y(n), y(n–1), …}, u(n) and y(n) being either uni or multidimensional signals. In the following, we consider filters having one input and one output. The generalization to multidimensional signals is straightforward.

There are two types of filters: (i) *transversal filter*s (termed Finite Impulse Response or FIR filters in linear filtering) whose outputs are functions of the input signals only; and (ii) *recursive filters* (termed Infinite Impulse Response or IIR filters in linear filtering) whose outputs are functions both of the input signals and of a delayed version of the output signals. Hence, a transversal filter is defined by:

$$y(n) = \Phi [u(n), u(n-1), ..., u(n-M+1)], \quad (1)$$

where M is the length of the finite memory of the filter, and a recursive filter is defined by

$$y(n) = \Phi [u(n), u(n-1), ..., u(n-M+1), y(n-1), y(n-2), ..., y(n-N)] \quad (2)$$

where N is the order of the filter.

The ability of a filter to perform the desired task is expressed by a criterion; this criterion may be either quantitative, e.g., maximizing the signal to noise ratio for spatial filtering [see for instance Applebaum and Chapman 1976], minimizing the bit error rate in data transmission [see for instance Proakis 1983], or qualitative, e.g. listening for speech prediction [see for instance Jayant and Noll 1984]. In practice, the criterion is usually expressed as a weighted sum of squared differences between the output of the filter and the desired output (e.g. LS criterion).

An *adaptive* filter is a system whose parameters are *continually* updated, without explicit control by the user. The interest in adaptive filters stems from two facts: (i) tailoring a filter of given architecture to perform a specific task requires a priori knowledge of the characteristics of the input signal; since this knowledge may be absent or partial, systems which can learn the characteristics of the signal are desirable; (ii) filtering nonstationary signals necessitates systems which are capable of tracking the variations of the characteristics of the signal.

The bulk of adaptive filtering theory is devoted to *linear* adaptive filters, defined by relations (1) and (2), where Φ is a linear function. Linear filters have been extensively studied, and are appropriate for many purposes in signal processing. A family of particularly efficient adaptation algorithms has been specially designed in the case of transversal linear filtering; they are referred to as the recursive least square (RLS) algorithms and their fast (FRLS) versions [Bellanger 1987, Haykin 1991].

Linear adaptive filters are widely used for system and signal modelling, due to their simplicity, and due to the fact that, in many cases (such as the estimation of gaussian signals), they are optimal. Despite their popularity, they remain inappropriate in many cases, especially for modelling non-linear systems; investigations along these lines have been performed for adaptive detection [see for instance Picinbono 1988], prediction and estimation [see for instance McCannon et al. 1982]. Unfortunately, when dealing with non-linear filters, no general adaptation algorithm is available, so that heuristic approaches are used.

By contrast, general methods for training neural networks are available; furthermore, neural networks are known to be universal approximants [see for instance Hornik et al. 1989], so that they can be used to approximate any smooth non-linear function. Since both the adaptation of filters [Haykin 1991, Widrow and Stearns 1985] and the training of neural networks involve gradient techniques, we propose to build on this algorithmic similarity a general framework which encompasses neural networks and filters. We do this in such a way as to clarify how neural networks can be applied to adaptive filtering problems.

1.2. NEURAL NETWORKS

The reader is assumed to be familiar with the scope and principles of the operation of neural networks; in order to help clarify the relations between neural nets and filters, the present section presents a broad classification of neural network architectures and functions, restricted to networks with supervised training.

1.2.1. - Functions of neural networks.

The functions of neural networks depend on the network architectures and on the nature of the input data:

- *network architectures:* neural networks can have either a feedforward structure or a feedback structure;
- *input data:* the succession of input data can be either time-ordered or arbitrarily ordered.

Feedback networks (also termed recurrent networks) have been used as associative memories, which store and retrieve either fixed points or trajectories in state space. The present paper stands in a completely different context: we investigate feedback neural networks which are never left to evolve under their own dynamics, but which are continually fed with new input data. In this context, the purpose of using neural networks is not that of storing and retrieving data, but that of capturing the (possibly non-stationary) characteristics of a signal or of a system.
Feedforward neural networks have been used basically as classifiers for patterns whose sequence of presentation is not significant and carries no information, although the ordering of components within an input vector may be significant.
In contrast, the time ordering of the sequence of input data is of fundamental importance for filters: the input vectors can be, for instance, the sequence of values of a sampled signal. At time n, the network is presented with a window of the last M values of the sampled signal {u(n), u(n-1), ..., u(n-M+1)}, and, at time n+1, the input is shifted by one time period {u(n+1), u(n), ..., u(n-M+2)}. In this context, *feedforward networks* are used as transversal filters, and *feedback networks* are used as recursive filters.

A very large number of examples of feedforward networks for classification can be found in the literature. Neural network associative memories have also been very widely investigated [Hopfield 1982, Personnaz 1986, Pineda 1987]. Feedforward networks have been used for prediction [Lapedes and Farber 1988, Pearlmutter 1989, Weigend et al. 1990]. Examples of feedback networks for filtering can be found in [Robinson and Fallside 1989, Elman 1990, Poddar and Unnikrishnan 1991].
Note that the above classification is not meant to be rigid. For instance, Chen et al. [Chen et al. 1990] encode a typical filtering problem (channel equalization) into a classification problem. Conversely, Waibel et al. [Waibel et al. 1989] uses a typical transversal filter structure as a classifier.

1.2.2. - Non-adaptive and adaptive training.

At present, in the vast majority of cases, neural networks are *not* used adaptively: they are first trained with a *finite* number of training samples, and subsequently used, e.g. for classification purposes. Similarly, non-adaptive filters are first trained with a finite number of time-ordered samples, and subsequently used with fixed coefficients. In contrast, *adaptive* systems are trained *continually* while being used with an *infinite* number of samples. The instances of neural networks being trained adaptively are quite few [Williams and Zipser 1989a, Williams and Zipser 1989b, Williams and Peng 1990, Narendra and Parthasarathy 1990, Narendra and Parthasarathy 1991].

2. STRUCTURE OF NEURAL NETWORKS FOR NON-LINEAR FILTERING

2.1. MODEL OF DISCRETE-TIME NEURON

The behaviour of a discrete-time neuron is defined by relation (3):

$$z_i(n) = f_i[v_i(n)] = f_i\left[\sum_{j\in P_i}\sum_{\tau=0}^{q_{ij}} c_{ij,\tau}\, z_j(n-\tau)\right] \qquad (3)$$

where:

f_i is the activation function of neuron i,
v_i is the potential of neuron i,
z_j can be either the output of neuron j or the value of a network input j,
P_i is the set of indices of the afferent neurons and network inputs to neuron i,
$c_{ij,\tau}$ is the weight of the synapse which transfers information from neuron or network input j to neuron i with (discrete) delay τ,
q_{ij} is the maximal delay between neuron j and neuron i.

It should be clear that several synapses can transfer information from neuron (or network input) j to neuron i, each synapse having its own delay τ and its own weight $c_{ij,\tau}$.
Obviously, one must have $c_{ii,0}=0\ \forall i$ for causality to be preserved.
If neuron i is such that: $i \notin P_i$ and $q_{ij}=0\ \forall j\in P_i$, neuron i is said to be static.

2.2. STRUCTURE OF NEURAL NETWORKS FOR FILTERING

The architecture of a network, i.e. the topology of the connections and the distribution of delays, may be fully or partially imposed by the problem that must be solved: the problem defines the sequence of input signal values and of desired outputs; in addition, a priori knowledge of the problem may give hints which help designing an efficient architecture (see for instance the design of the feedforward network described in [Waibel et al.1989]).
In order to clarify the presentation and to make the implementation of the training algorithms easier, the *canonical form* of the network is especially convenient. We first introduce the canonical form of feedback networks; the canonical form of feedforward networks will appear as a special case.

2.2.1. - The canonical form of feedback networks

The dynamics of a discrete-time feedback network can be described by a finite-difference equation of order N, which can be expressed by a set of N first-order difference equations involving N variables (termed state variables) in addition to the M input variables. *Thus, any feedback network can be cast into a canonical form which consists of a feedforward (static) network*

- *whose outputs are the outputs of the neurons which have desired values, and the values of the state variables,*
- *whose inputs are the inputs of the network and the values of the state variables, the latter being delayed by one time unit* (Figure 1).

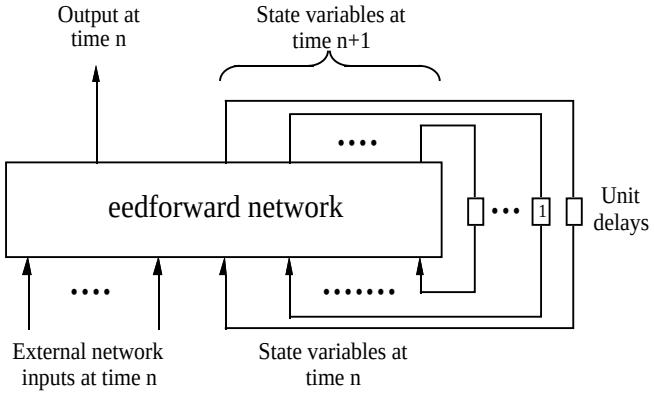

Figure 1:
General canonical form of a feedback neural network.

Note that the choice of the set of state variables is not necessarily unique: therefore, a feedback network may have several canonical forms. The *state of the network* is the set of values of the state variables.

In the following, all vectors will be denoted by uppercase letters.

The behaviour of a single-input-single-output network is described by the state equation (4) and output equation (4'):

$S(n+1) = \varphi[S(n),U(n)]$ (4)

$y(n) = \psi[S(n),U(n)]$ (4')

where U(n) is the vector of the M last successive values of the external input u and S(n) is the vector of the N state variables (state vector). The output of the network may be a state variable.

The transformation of a non-canonical feedback neural network filter to its canonical form requires the determination of M and of N. In the single-input-single-output case, the computation of the maximum number of external inputs E (M≤E) is done as follows: construct the *network graph* whose nodes are the neurons and the input, and whose edges are the connections between neurons, weighted by the values of the delays; find the direct path of maximum weight D from input to output; one has E = D+1. The determination of the order N of the network from the network graph is less straightforward; it is described in Appendix 1.

If the task to be performed does not suggest or impose any structure for the filter, one may use either a multi-layer Perceptron, or the most general form of feedforward network in the canonical form, i.e. a fully connected network; in that case, the number of neurons, of state variables and of delayed inputs must be found by trial and error.

If we assume that the state variables are delayed values of the output, or if we assume that the state of the system can be reconstructed from values of the input and output, then all state variables have desired values. Such is the case for the NARMAX model [Chen and Billings 1989] and for the systems investigated in [Narendra 1990]. Figure 2 illustrates the most general form of the canonical form of a network having a single output y(n) and N state variables {y(n-1), ..., y(n-N+1)}. It features M external inputs, N feedback inputs and one output; it can implement a fairly large class of functions Φ; the non-recursive part of the network (which implements function Φ) is a fully-connected feedforward net.

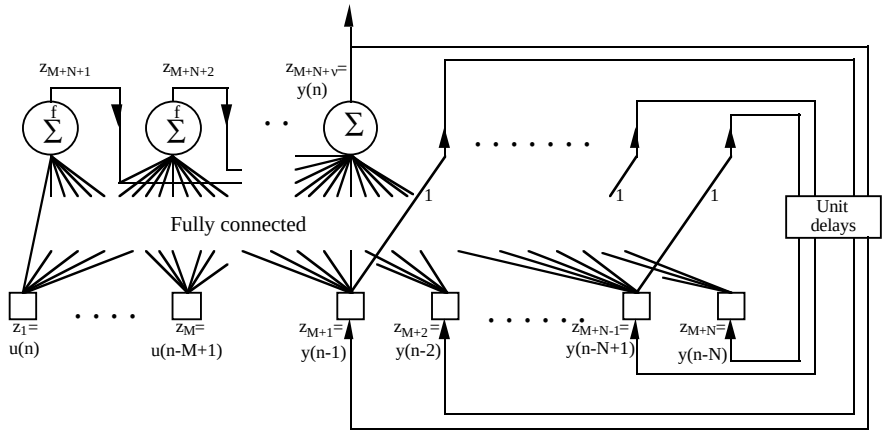

Figure 2:
Canonical form of a network with a fully-connected feedforward net,
whose state variables are delayed values of the output.

More specific architectures are described in the literature, implementing various classes of functions φ and ψ. Some examples of such architectures are presented in Appendix 2.

2.2.2. - Special case: the canonical form of feedforward networks

Similarly, any feedforward network with delays, with input signal u, can be cast into the form of a feedforward network of static neurons, whose inputs are the successive values u(n), u(n-1), ..., u(n-M+1); this puts the network under the form of a transversal filter obeying relation (1):
$y(n) = \Phi [u(n), u(n-1), ..., u(n-M+1)] = \Phi [U(n)]$.
The transformation of a non-canonical feedforward neural network filter to its canonical form requires the determination of the maximum value M, which is done as explained above in the case of feedback networks. An example described in Appendix 1 shows that this transformation may introduce the replication of some weights, known as "shared weights".

3. TRAINING ADAPTIVE NEURAL NETWORKS FOR ADAPTIVE FILTERING

3.1. CRITERION

The task to be performed by a neural network used as a filter is defined by a (possibly infinite) sequence of inputs u and of corresponding desired outputs d. At each sampling time n, an error e(n) is defined as the difference between the desired output d(n) and the actual output of the network y(n): e(n)=d(n)-y(n). For instance, in a modelling process, d(n) is the output of the process to be modelled; in a predictor, d(n) is the input signal at time n+1.
The training algorithms aim at finding the network coefficients so as to satisfy a given quality criterion. For example, in the case of non-adaptive training (as defined in Section 1.2.2), the most popular criterion is the Least Squares (LS) criterion; the cost function to be minimized is

$$J(C) = \frac{1}{K} \sum_{p=1}^{K} e(p)^2$$

Thus, the coefficients minimizing J(C) are first computed with a finite number K of samples; the network is subsequently used with these fixed coefficients.
In the context of adaptive training, taking into account all the errors since the beginning of the optimization does not make sense; thus, one can implement a forgetting mechanism. In the present paper, we use a rectangular "sliding window" of length N_C; hence the following cost function:

$$J(n, C) = \frac{1}{2} \sum_{p=n-N_c+1}^{n} e(p)^2 .$$

The choice of the length N_C of the window is task-dependent, and is related to the typical time scale of the non-stationarity of the signal to be processed.
In the following, the notation J(n) will be used instead of J(n, C). The computation of e(p) will be discussed in sections 3.3 and 3.4.2.

3.2. ADAPTIVE TRAINING ALGORITHMS

Adaptive algorithms compute, in real time, coefficient modifications based on past information. In the present paper, we consider only gradient-based algorithms, which require the estimation of the gradient of the cost function, $\nabla J(n)$, and possibly the estimation of J(n); these computations make use of data available at time n.
In the simplest and most popular formulation, a single modification of the vector of coefficients $\Delta C(n)=C(n)-C(n-1)$ is computed between time n and time n+1; such a method, usual in adaptive filtering, is termed a *purely recursive* algorithm.
The modification of the coefficients is often performed by the steepest-descent method, whereby $\Delta C(n)=-\mu\nabla J(n)$. In order to improve upon the steepest-descent method, quasi-Newton methods can be used [Press et al. 1986], whereby $\Delta C(n)=+\mu D(n)$, where D(n) is a vector obtained by a linear transformation of the gradient.
Purely recursive algorithms were introduced in order to avoid time-consuming computations between the reception of two successive samples of the input signal. If the application under investigation does not have stringent time requirements, then other possibilities can be considered. For instance, if it is desired to get closer to the minimum of the cost function, several iterations of the gradient algorithm can be performed between time n and time n+1. In that case, the coefficient-modification vector $\Delta C(n)$ is computed iteratively as:
$\Delta C(n) = C_{K_n}(n) - C_0(n)$ where K_n is the number of iterations at time n,
with $C_k(n) = C_{k-1}(n) + \mu_k D_{k-1}(n)$ (k=1 to K_n), where $D_{k-1}(n)$ is obtained from the coefficients computed at iteration k-1,
and $C_0(n) = C_{K_{n-1}}(n-1)$.
If $K_n>1$, the tracking capabilities of the system in the non-stationary case, or the speed of convergence to a minimum in the stationary case, may be improved with respect to the purely recursive algorithm. The applicability of this method depends specifically on the ratio of the typical time scale of the non-stationarity to the sampling period.
As a final variant, it may be possible to update the coefficients with a period T>1 if the time scale of the non-stationarity is large with respect to the sampling period:
$C_0(n) = C_{K_{n-T}}(n-T)$.

Whichever algorithm is chosen, the central problem is the estimation of the gradient, $\nabla J(n)$:

$$\frac{\partial J(n)}{\partial c_{ij}} = \frac{\partial}{\partial c_{ij}} \left(\frac{1}{2} \sum_{p=n-N_c+1}^{n} e(p)^2 \right).$$

At present, two techniques are available for this computation: the forward computation of the gradient and the popular backpropagation of the gradient.
i) The forward computation of the gradient is based on the following relation:

$$\frac{\partial J(n)}{\partial c_{ij}} = - \sum_{p=n-N_c+1}^{n} e(p) \frac{\partial y(p)}{\partial c_{ij}} .$$

The partial derivatives of the output at time n with respect to the coefficients appearing on the right-hand side are computed recursively in the forward direction, from the partial derivatives of the inputs to the partial derivatives of the outputs of the network.

ii) In contrast, backpropagation uses a chain derivation rule to compute the gradient of J(n). The required partial derivatives of the cost function J(n) with respect to the potentials are computed in the backward direction, from the output to the inputs.

The advantages and disadvantages of these two techniques will be discussed in sections 3.3 and 3.4.2.

In the following, we show how to compute the coefficient modifications for feedforward and feedback neural networks, and we put into perspective the training algorithms developed recently for neural networks and the algorithms used classically in adaptive filtering.

3.3. TRAINING FEEDFORWARD NEURAL NETWORKS FOR NON-LINEAR TRANSVERSAL ADAPTIVE FILTERING.

We consider purely recursive algorithms (i.e. T=1 and K_n=1).The extension to non-purely recursive algorithms is straightforward.

As shown in section 2.2.2, any discrete-time feedforward neural network can be cast into a canonical form in which all neurons are static. The output of such a network is computed from the M past values of the input, and the output at time n does not depend on the values of the output at previous times.

Therefore, the cost function

$$J(n) = \frac{1}{2} \sum_{p=n-N_c+1}^{n} e(p)^2$$

is a sum of N_C *independent* terms. Its gradient can be computed, from the N_C+M+1 past input data and the N_C corresponding desired outputs, as a sum of N_C independent terms: therefore, the modification of the coefficients, at time n, is the sum of N_C elementary modifications computed from N_C independent, *identical* elementary *blocks* (each of them with coefficients C(n-1)), *between time n and time n+1*.

We introduce the following notation, which will be used both for feedforward and for feedback networks: the blocks are numbered by m; all values computed from block m of the training network will be denoted with superscript m. For instance, $y^m(n)$ is the output value of the network computed by the m-th block *at time n*: it is the value that the output of the filter would have taken on, at time n-N_C+m, if the vector of coefficients of the network at that time had been equal to C(n-1).

With this notation, the cost function taken into account for the modification of the coefficients at time n becomes:

$$J(n) = \frac{1}{2} \sum_{m=1}^{N_c} [e^m(n)]^2 \quad \text{where } e^m(n) = d(n-N_c+m) - y^m(n) \text{ is the error for block m computed at time n.}$$

As mentioned in section 3.2, two techniques are available for computing the gradient of the cost function: the forward computation technique (used classically in adaptive filtering) and the backpropagation technique (used classically for neural networks) [Rumelhart et al. 1986].

Thus, each block, from block m=1 to block m=N_C , computes a partial modification Δc_{ij}^m of the coefficients and the total modification, at time n, is:

$$\Delta c_{ij}(n) = \sum_{m=1}^{N_c} \Delta c_{ij}^m(n) ,$$

as illustrated in Figure 3.

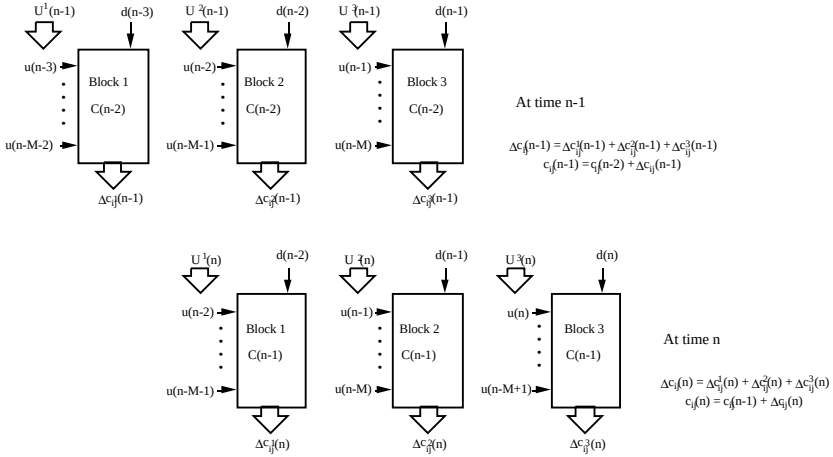

Figure 3:
Computation of two successive coefficient modifications
for a non-linear transversal filter (N_c=3).

It was mentioned above that either the forward computation method or the backpropagation method can be used for the estimation of the gradient of the cost function. Both techniques lead to exactly the same numerical results; it has been shown [Pineda 1989] that backpropagation is less computationally expensive than forward computation. Therefore, *for the training of feedforward networks operating as non-linear transversal filters*, backpropagation is the preferred technique for gradient estimation. However, as we shall see in the following, this is not always the case *for the training of feedback networks*.

3.4. TRAINING FEEDBACK NEURAL NETWORKS FOR NON-LINEAR RECURSIVE ADAPTIVE FILTERING.

This section is devoted to the adaptive training of feedback networks operating as recursive filters. This problem is definitely richer, and more difficult, than the training of feedforward networks for adaptive transversal filtering. We present a wide variety of algorithms, and elucidate their relationships to adaptation algorithms used in linear adaptive filtering and to neural network training algorithms.

3.4.1. - General presentation of the algorithms for training feedback networks:

Since the state variables and the output of the network at time n depend on the values of the state variables of the network at time n-1, the computation of the gradient of the cost function requires the computation of partial derivatives from time n=0 up to the present time n. This is clearly not practical, since (i) the amount of computation would grow without bound, and (ii) in the case of non-stationary signals, taking into account the whole past history does not make sense. Therefore, the estimation of the gradient of the cost function is performed by truncating the computations to a fixed number of sampling periods N_t into the past. Thus, one has to use N_t computational blocks (defined below), numbered from m=1 to m=N_t : the outputs $y^m(n)$ are computed through N_t identical versions of the feedforward part of the canonical form of the network (each of them with coefficients C(n-1)). Clearly, N_t must be larger than or equal to N_C in order to compute the N_C last errors $e^m(n)$.
Here again, we first consider the case where T=1 and K_n=1.
Figure 4 shows the m-th computational block for the *forward computation* technique: the state input vector is denoted by $S_{in}^m(n)$; the state output vector is denoted by $S_{out}^m(n)$. The canonical feedforward (FF) net computes the output from the external inputs $U^m(n)$ and the state inputs $S_{in}^m(n)$. The Forward Computation (FC) net computes the partial derivatives required for the coefficient modification, and the partial derivatives of the state vector which may be used by the next block. The N_t blocks compute sequentially the N_t outputs $\{y^m\}$ and the partial derivatives $\{\partial y^m/\partial c_{ij}\}$, in the forward direction (m=1 to N_t). The N_C errors $\{e^m\}$ (computed from the outputs of the last N_C blocks), and the corresponding partial derivatives are used for the computation of the coefficient modifications, which is the sum of N_C terms:

$$\Delta c_{ij}(n) = -\mu \frac{\partial J(n)}{\partial c_{ij}} = \mu \sum_{m=N_t-N_c+1}^{N_t} e^m \frac{\partial y^m}{\partial c_{ij}} = \sum_{m=N_t-N_c+1}^{N_t} \Delta c_{ij}^m(n) \ .$$

Details of the computations are to be found in Appendix 3.

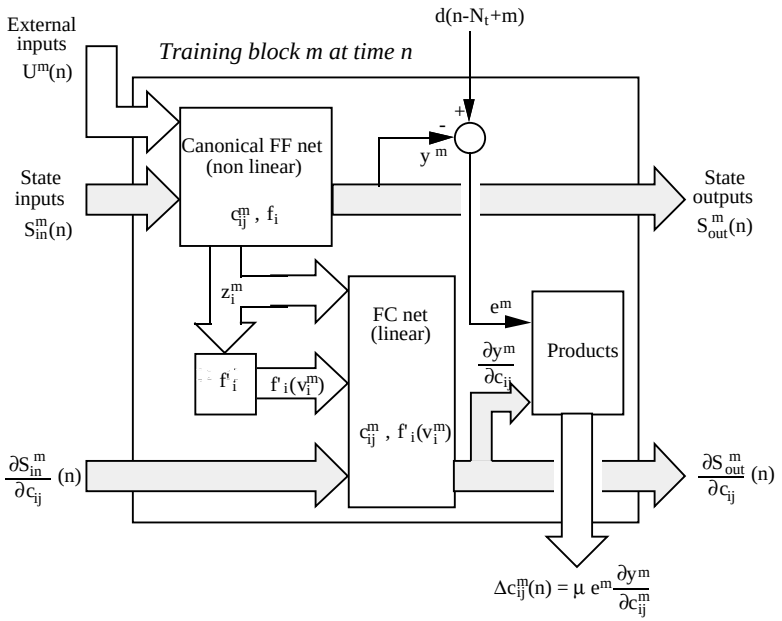

Figure 4:
Training block m at time n with a desired output value: computation of a partial coefficient modification using the forward computation of the gradient for a feedback neural network. If the ouput of block m has no desired value, it has no "products" part and does not contribute directly to coefficient modifications: it just transmits the state variables and their derivatives to the next block.

In order for the blocks to be able to perform the above computations, the values of the state inputs $S_{in}^m(n)$ and of their partial derivatives with respect to the weights must be determined. The choice of these values is of central importance; it gives rise to four families of algorithms.

3.4.2. - Choice of the state inputs and of their partial derivatives.

3.4.2.1. - Choice of the state inputs:

The most "natural "choice of the state inputs of block m is to take the values of the state variables computed by block m-1: $S_{in}^{m}(n)=S_{out}^{m-1}(n)$ with $S_{in}^{1}(n)=S_{out}^{1}(n-1)$. Thus, the trajectory of the network in state space, computed at time n, is independent of the trajectory of the process: the input of block m is not directly related to the actual values of the state variables of the process to be modelled by the network, hence the name **undirected algorithm.** If the coefficients are mismatched, this choice may lead to large errors and to instabilities. Figure 5a shows pictorially the desired trajectory of the state of the network and the trajectory which is computed at time n when an undirected algorithm is used (N_t=3, N_c=2). We show in section 3.4.2.2 that, in that case, one must use the forward computation technique to compute the coefficient modifications (Figure 5b).

This choice of the state inputs has been known as the *output error* approach in adaptive filtering and as the *parallel* approach in automatic control. It does not require that all state variables have desired values.

In order to reduce the risks of instabilities, an alternative approach may be used, called a **semi-directed algorithm**. In this approach, the state of the network is constrained to be identical to the desired state for m=1:

$S_{in}^{m}(n)=S_{out}^{m-1}(n)$ with $S_{in}^{1}(n) = [d(n-N_t), d(n-N_t-1), ..., d(n-N_t-M+1)]$. This is possible only when the chosen model is such that desired values are available for all state variables; this is the case for the NARMAX model. Figure 6a shows pictorially the desired trajectory of the state of the network and the trajectory which is computed at time n when a semi-directed algorithm is used (N_t=4, N_c=2). We show in section 3.4.2.2 that, in that case, one can use the backpropagation technique to compute the coefficient modifications (Figure 6b).

The trajectory of the state of the network can be further constrained by choosing the state inputs *of all blocks* to be equal to their desired values:

$S_{in}^{m}(n) = [d(n-N_t+m-1), d(n-N_t+m-2), ..., d(n-N_t+m-M)]$ for all m.

With this choice, the training is under control of the desired values, hence of the process to be modelled, at each step of the computations necessary for the adaptation (hence the name **directed algorithm**); therefore, it can be expected that the influence of the mismatch of the model to the process is less severe than in the previous cases. Figure 7a shows pictorially the desired trajectory of the state of the network and the trajectory which is computed at time n when a directed algorithm is used (N_t= N_c=3). We show in section 3.4.2.2 that, in that case, one can use the backpropagation technique to compute the coefficient modifications (Figure 7b). In directed algorithms, all blocks are independent, just as in the case of the training of feedforward networks (section 3.3); therefore, one has $N_t = N_c$.

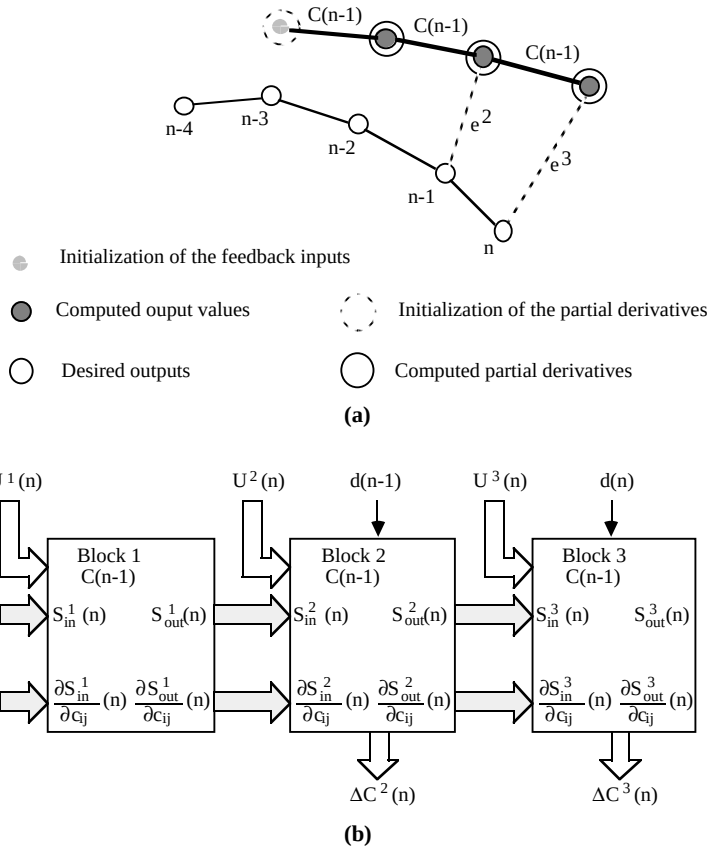

Figure 5:
Undirected algorithm (with N_t=3 and N_c=2).
a) Pictorial representation of the desired trajectory, and of the trajectory computed at time n, in state space; the trajectory at time n is computed by the blocks shown on Figure 5b.
b) Computational system at time n. The detail of each block is shown on Figure 4. Note that the output of block 1 has no desired value.

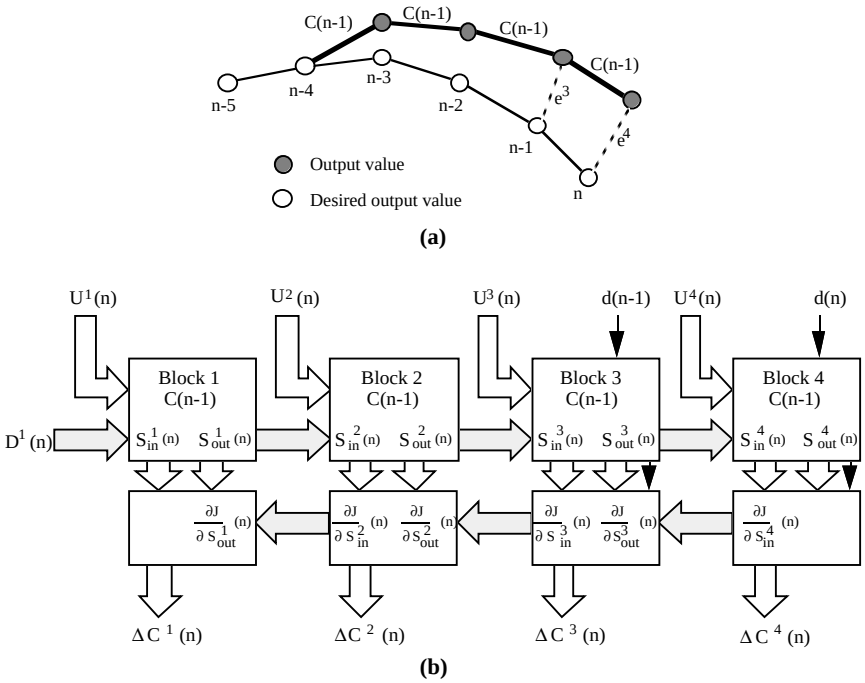

Figure 6:
Semidirected algorithm (with N_t=4 and N_C=2).
a) Pictorial representation of the desired trajectory, and of the trajectory computed at time n, in state space; the trajectory at time n is computed by the blocks shown on Figure 6b.
b) Computational system at time n. The detail of each block is shown on Figure 8. Note that the outputs of blocks 1 and 2 have no desired values, but do contribute an additive term to the coefficient modifications.

This choice of the values of the state inputs has been known as the *equation error* approach in adaptive filtering and as the *series-parallel* approach in automatic control. It is an extension of the *teacher forcing* technique [Jordan 1985] used for neural network training.

If some state inputs do not have desired values, **hybrid** versions of the above algorithms can be used: those state inputs for which no desired values are available are taken equal to the corresponding computed state variables (as in an undirected algorithm), whereas the other state inputs may be taken equal to their desired values (as in a directed or in a semi-directed algorithm).

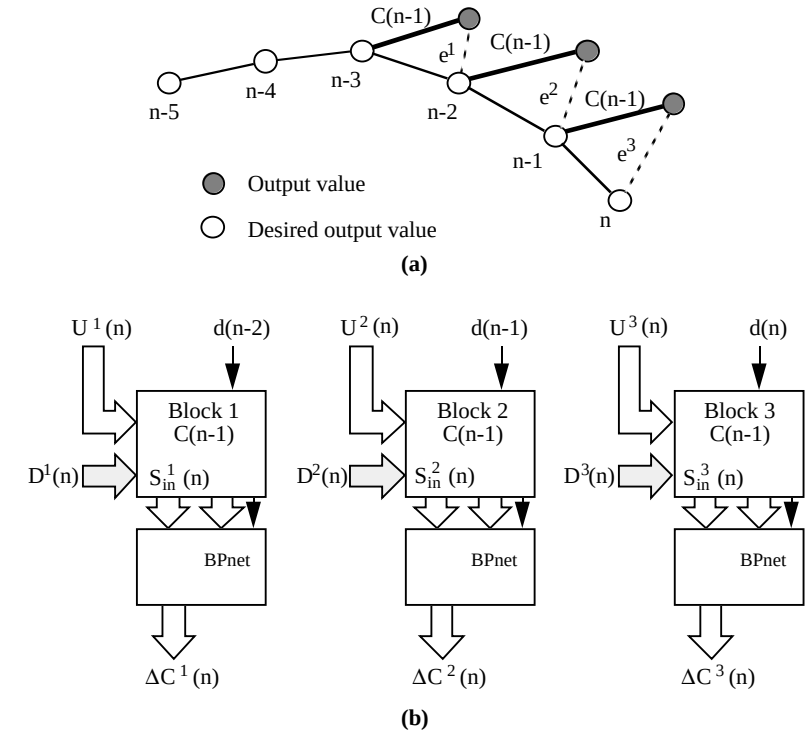

Figure 7:
Directed algorithm (with N_t=N_C=3).
a) Pictorial representation of the desired trajectory, and of the trajectory computed at time n, in state space; the trajectory at time n is computed by the blocks shown on Figure 7b.
b) Computational system at time n. The detail of each block is shown on Figure 8.
Note that, in a directed algorithm, each block is independent from the others and must have a desired output value.

3.4.2.2. - Consistent choices of the partial derivatives of the state inputs:

The choices of the state inputs lead to corresponding choices for the initialization of the partial derivatives, as illustrated in Figures 5a, 6a, 7a.

In the case of the *undirected* algorithm, one has $S_{in}^{m}(n)=S_{out}^{m-1}(n)$; therefore, a consistent choice of the values of the partial derivatives of the state inputs consists in taking the values of the partial derivatives of the state outputs computed by the previous block:

$$\frac{\partial S_{in}^{m}(n)}{\partial c_{ij}} = \frac{\partial S_{out}^{m-1}(n)}{\partial c_{ij}},$$

except for the first block where one has:

$$\frac{\partial S_{in}^{1}(n)}{\partial c_{ij}} = \frac{\partial S_{out}^{1}(n-1)}{\partial c_{ij}}.$$

In the case of the *semi-directed* algorithm, the state input values of the first block are taken equal to the corresponding desired values; the latter do not depend on the coefficients; therefore, their partial derivatives can consistently be taken equal to zero. The values of the partial derivatives of the state inputs are taken equal to the values of the partial derivatives of the state outputs computed by the previous block.

In the case of the *directed* algorithm, one can consistently take the partial derivatives of the state inputs of all blocks equal to zero.

The parameters T, K_n, N_t, N_c being fixed, the first three algorithms described above are summarized on the first line of each section of Table 1. The first part of the acronyms refers to the choice of the state inputs and the second part refers to the choice of the partial derivatives of the state inputs. They include algorithms which have been used previously by other authors: the *"Real-Time Recurrent Learning Algorithm"* [Williams and Zipser 1989a] is an undirected algorithm (using the forward computation technique) with $N_t=N_c=1$. This algorithm is known as the Recursive Prediction Error algorithm, or IIR-LMS algorithm, in linear adaptive filtering [Widrow and Stearns 1985]. The *"Teacher-Forced Real-Time Recurrent Learning Algorithm"* [Williams and Zipser 1989a] is a hybrid algorithm with $N_t=N_c=1$.

	Initialization: state input of the first block $S_{in}^{1}(n) =$	**State input of a current block** $S_{in}^{m}(n) =$	**Initialization: partial derivatives for the first block** $\frac{\partial S_{in}^{1}}{\partial c_{ij}}(n) =$	**Partial derivatives for a current block** $\frac{\partial S_{in}^{m}}{\partial c_{ij}}(n) =$
Undirected (UD) Algorithm (Output Error) (Parallel)	$S_{out}^{1}(n-1)$	$S_{out}^{m-1}(n)$	$\frac{\partial S_{out}^{1}}{\partial c_{ij}}(n-1)$	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$
UD-D Algorithm	$S_{out}^{1}(n-1)$	$S_{out}^{m-1}(n)$	zero	zero
UD-SD Algorithm	$S_{out}^{1}(n-1)$	$S_{out}^{m-1}(n)$	zero	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$

	Initialization: state input of the first block $S_{in}^{1}(n) =$	**State input of a current block** $S_{in}^{m}(n) =$	**Initialization: partial derivatives for the first block** $\frac{\partial S_{in}^{1}}{\partial c_{ij}}(n) =$	**Partial derivatives for a current block** $\frac{\partial S_{in}^{m}}{\partial c_{ij}}(n) =$
Semi-Directed (SD) Algorithm	Desired values	$S_{out}^{m-1}(n)$	zero	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$
SD-D Algorithm	Desired values	$S_{out}^{m-1}(n)$	zero	zero
SD-UD Algorithm	Desired values	$S_{out}^{m-1}(n)$	$\frac{\partial S_{out}^{1}}{\partial c_{ij}}(n-1)$	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$

	Initialization: state input of the first block $S_{in}^{1}(n) =$	**State input of a current block** $S_{in}^{m}(n) =$	**Initialization: partial derivatives of the first block** $\frac{\partial S_{in}^{1}}{\partial c_{ij}}(n) =$	**Partial derivatives for a current block** $\frac{\partial S_{in}^{m}}{\partial c_{ij}}(n) =$
Directed Algorithm (D) (Equation Error) (Teacher Forcing) (Series Parallel)	Desired values	Desired values	zero	zero
D-SD Algorithm	Desired values	Desired values	zero	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$
D-UD Algorithm	Desired values	Desired values	$\frac{\partial S_{out}^{1}}{\partial c_{ij}}(n-1)$	$\frac{\partial S_{out}^{m-1}}{\partial c_{ij}}(n)$

Table 1:
Three families of algorithms for the training of feedback neural networks. In each section, the first line describes the algorithms with consistent choices of the state inputs (sec. 3.4.2.2).

The above algorithms have been introduced in the framework of the *forward computation* of the gradient of the cost function. However, the estimation of the gradient of the cost function by *backpropagation* is attractive with respect to computation time, as mentioned in section 3.3.4. If this technique is used, the computation is performed with N_t blocks, where each coefficient c_{ij} is replicated in each block m as c_{ij}^m. Therefore, one has:

$$\frac{\partial v_i^m}{\partial c_{ij}^m} = z_j^m \ .$$

The training block m is shown in Figure 8: after computing the N_C errors using the N_t blocks in the forward direction, the N_t blocks compute the derivatives of J(n) with respect to the potentials $\{v_i^m\}$, in the backward direction. The modification of the coefficients is computed from the N_t blocks as:

$$\Delta c_{ij}(n) = - \mu \frac{\partial J(n)}{\partial c_{ij}} = \mu \sum_{m=1}^{N_t} \frac{\partial J(n)}{\partial v_i^m} z_j^m = \sum_{m=1}^{N_t} \Delta c_{ij}^m(n) \ .$$

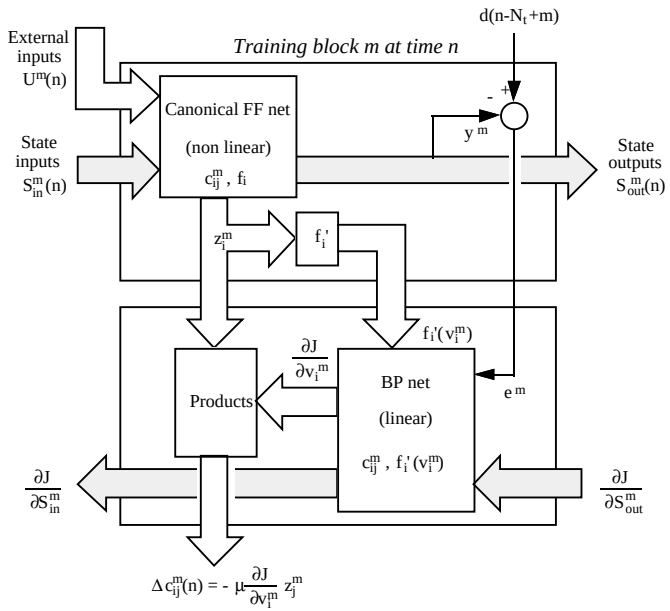

Figure 8:
Training block m at time n with a desired output value: computation of a partial coefficient modification using the backpropagation technique for the estimation of the gradient for a feedback neural network. If block m has no desired value, then $e^m=0$, but it does contribute an additive term to the coefficient modification. It should be noticed that forward propagation through all blocks must be performed before backpropagation.

It is important to notice that backpropagation assumes implicitly that the partial derivatives of the state inputs of the first copy are taken equal to zero. Therefore, the backpropagation technique will lead to the same coefficient modifications as the forward propagation technique if and only if it is used within algorithms complying with this condition, i.e. within directed or semi-directed algorithms (Figures 6b and 7b); *backpropagation cannot be used consistently within undirected and hybrid algorithms.* When both backpropagation and forward computation techniques can be used, backpropagation is the best choice because of its lower computational complexity.

An example of the use of a directed algorithm for idenfication and control of non-linear processes can be found in [Narendra and Parthasarathy 1990].

3.4.2.3. - Other choices of the partial derivatives of the state inputs:

Because adaptive neural networks require real-time operation, tradeoffs between consistency and computation time may be necessary: setting partial derivatives $\partial S_{in}^m/\partial c_{ij}$ equal to zero may save time by making the computation by backpropagation possible even for undirected algorithms (UD-D or UD-SD algorithms). The full variety of algorithms is shown on Table 1: in each group, the first line shows the characteristics of the fully consistent algorithm, whereas the other two lines show other possibilities which are not fully consistent, but which can nevertheless be used with advantage. The SD-UD, D-SD and D-UD algorithms have been included for completeness: computation time permitting, the accuracy of the computation may be improved by setting the partial derivatives of the state inputs to non-zero values in the directed or semi-directed case.

Undirected algorithms have been in use in linear adaptive filtering: the *extended LMS* algorithm is a UD-D algorithm (see table 1) with $N_t=N_C=1$ [Shynk 1989]; the *a posteriori error* algorithm is also a UD-D algorithm with $N_t=2$, $N_C=1$ [Shynk 1989].

The *truncated backpropagation through time* algorithm [Williams and Peng 1990] is a UD-D algorithm with $N_C=1$ and $N_t>1$, with a special feature: in order to save computation time, the coefficients of the blocks 1 to N_t-1 are the coefficients which were computed at the corresponding times.

CONCLUSION

The present paper provides a comprehensive framework for the adaptive training of neural networks, viewed as non-linear filters, either transversal or recursive. We have introduced the concept of canonical form of a neural network, which provides a unifying view of network architectures and allows a general description of training methods based on gradient estimation. We have shown that backpropagation is always advantageous for training feedforward networks adaptively, but that it is not necessarily the best method for training feedback networks. In the latter case, four families of training algorithms have been proposed; some of these algorithms have been in use in classical linear adaptive filtering or adaptive control, whereas others are original.

The unifying concepts thus introduced are helpful in bridging the gap between neural networks and adaptive filters. Furthermore, they raise a number of challenging problems, both for basic and for applied research. From a fundamental point of view, general approaches to the convergence and stability of these algorithms are still lacking; a preliminary study along these lines has been presented [Dreyfus et al. 1992]; from the point of view of applications, the real-time operation of non-linear adaptive systems requires specific silicon implementations, thereby raising the questions of the speed and accuracy required for the computations.

ACKNOWLEDGEMENTS

The authors are very grateful to O. MACCHI for numerous discussions which have been very helpful in putting neural networks into the perspective of adaptive filtering. C. VIGNAT has been instrumental in formalizing some computational aspects of this work. We thank H. GUTOWITZ for his critical reading of the manuscript.

APPENDIX 1

We consider a discrete-time neural network with any arbitrary structure, and its associated network graph as defined in section 2.2.

The set of state variables is the minimal set of variables which must be initialized in order to allow the computation of the state of all neurons at any time n>0, given the values of the external inputs at all times from 0 to n. The order of the network is the number of state variables.

Clearly, the only neurons whose state must be initialized are the neurons which are within loops (i.e. within cycles in the network graph). Therefore, in order to determine the order of the network, the network graph should be pruned by suppressing all external inputs and all edges which are not within cycles (this may result in a disconnected graph).

To determine the order, it is convenient to further simplify the network graph as follows: (i) merge parallel edges into a single edge whose delay is the maximum delay of the parallel edges; (ii) if two edges of a loop are separated by a neuron which belongs to this loop only, suppress the neuron and merge the edges into a single edge whose delay is the sum of the delays of the edges.

We now consider the neurons which are still represented by nodes in the simplified network graph. We denote by N the order of the network.

If, for each node i of the simplified graph, we denote by A_i the delay of the synpase, afferent to neuron i, which has the largest delay (i.e. the weight of the edge directed towards i which has the largest weight), then a simple upper bound for N is given by:

$N \leq \sum_i A_i$.

The state x_i of a neuron i which has an afferent synapse of delay A_i cannot be computed at times $n<A_i$; the computation of the states of the other neurons may require the values of x_i at times 0, 1, ..., A_i-1; thus, the contribution of neuron i to the order of the network is smaller than or equal to A_i.

Let the quantity ω_i be defined as:

$\omega_i = A_i - min_{j \in R_i} (A_j - \tau_{ji})$ *if* $A_i - min_{j \in R_i} (A_j - \tau_{ji}) > 0$,

$\omega_i = 0$ *otherwise,*

where R_i stands for the set of indices of the nodes which are linked to i by an edge directed from i to j (i.e. the set of neurons to which neuron i projects efferent synapses).

Then the order of the network is given by:

$N = \sum_i \omega_i$.

The necessity of imposing the state of neuron i at time k ($0<k<A_i$-1) depends on whether this value is necessary for the computation of the state of a neuron j to which neuron i sends its state: if $k+\tau_{ji}$ is smaller than the maximum delay A_j of the synapses afferent to j, it is not necessary to transmit the state of neuron i at time k to neuron j, since the latter does not have the information required to

compute its state at time $k+\tau_{ji}$; the information on the state of neuron i at time k is necessary only if one has $k \geq A_j - \tau_{ji}$.

Therefore, the minimum number of successive values required for neuron i is equal to:

$A_i - \min_{j \in R_i} (A_j - \tau_{ji})$ if $A_i - \min_{j \in R_i} (A_j - \tau_{ji}) > 0$, zero otherwise.

Clearly, this result is in accord with the upper bound given above.

The above results determine the number of state variables related to each neuron. The choice of the set of state variables is not unique. The presence of parallel edges within a loop, or the presence of feedforward connections between loops, may require the replication of some neurons and of some coefficients.

Figure A1.1.a shows a feedback network and Figure A1.1.b shows its canonical form; the order of the network is 6. The example shows that some weights are replicated.

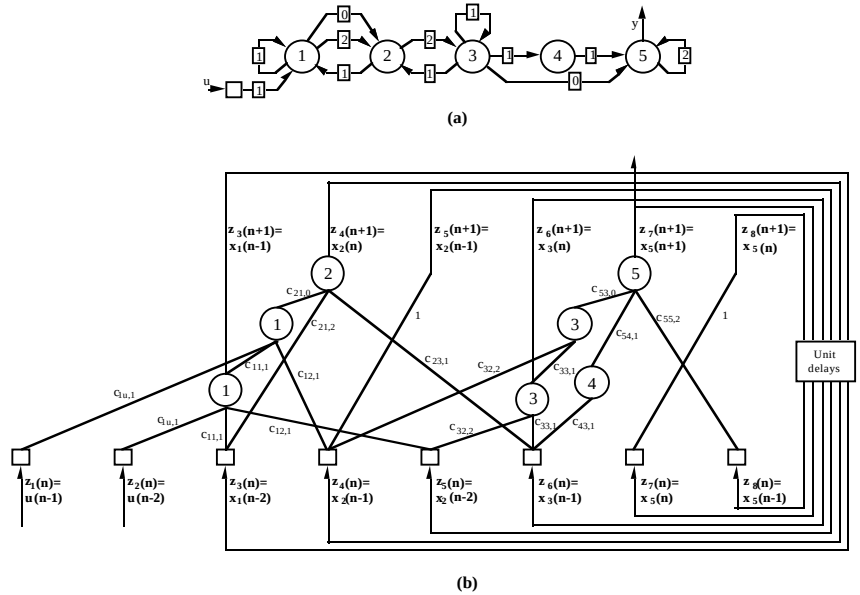

Figure A1.1:
a) Example of a feedback neural network. Numbers in rectangles are synapse delay values, u is the external input and y is the output of the network.
b) Canonical form of the network (E=8, M=2, N=6). The $c_{ij,\tau}$ notation of relation (3) is used.

APPENDIX 2

This appendix describes several architectures of feedback neural networks which have been proposed in the literature. We present their canonical form, so that they can be easily compared.

The discrete-time mathematical model of a time-invariant dynamical process is of the form

$S(n+1) = \varphi [S(n), U(n)]$

$Y(n) = \psi [S(n), U(n)]$,

where vector U is the input of the dynamical system, vector S denotes the state of the system, and vector Y is the output of the system. Since neural networks with hidden neurons are able to approximate a large class of non-linear functions, they can be used for implementing functions φ and ψ.

The network proposed by Jordan [Jordan 1986] is trained to produce a given sequence y(n) for a given constant input P ("plan"). Thus it is used as an associative memory. The network and its canonical form are shown in Figure A2.1. The representation of the network under its canonical form shows that the network is of order 2, although the representation used by Jordan exhibits four connections with unit delays. Note that the state variables are not delayed values of the output. The presence of hidden neurons allows this network to learn any function $y(n)=\psi[S(n), U(n)]$.

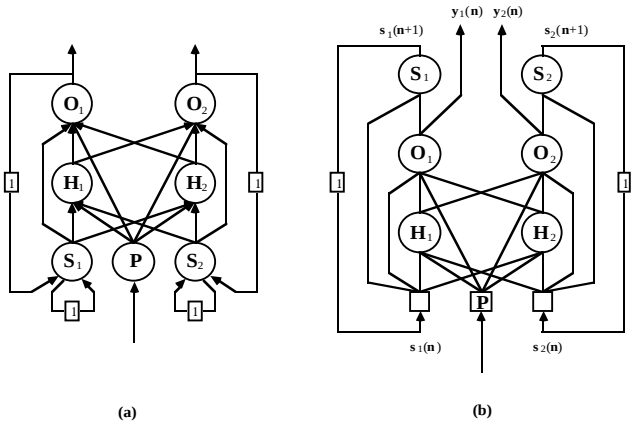

Figure A2.1:
a) Network architecture proposed by Jordan.
b) Canonical form.

The network suggested by Elman [Elman 1988] is used as a non-linear filter. Its canonical form is shown on Figure A2.2.

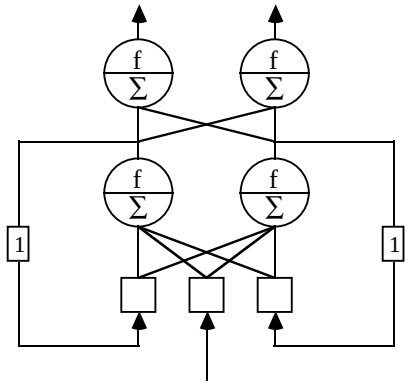

Figure A2.2:
Canonical form of the network architecture proposed by Elman.

Each state variable is computed as a fixed non-linear function f of a weighted sum of the external inputs and state inputs. Therefore, the class of functions φ which can be implemented is restricted to the form:
$\varphi[S(n),U(n)] = f[AS(n) + BU(n)]$ where A and B are the synaptic matrices.
Similarly, the output is computed as a fixed non-linear function f of a weighted sum of the state variables, so that the class of functions ψ that can be implemented is restricted to:
$\psi[S(n), U(n)] = f[CS(n)]$ where C is the synaptic matrix.

The network proposed in [Williams and Zipser 1989a, Williams and Peng 1990] is used as a non-linear filter. The state of the network at time n+1 is computed as a weighted sum of the inputs and of the state values at time n, followed by a fixed non-linearity f_i. As a result, the network can only implement non-linear functions of the form $f_i(AS(n) + BU(n))$.

The network used by Poddar and Unnikrishnan [Poddar and Unnikrishnan 1991] consists of a "feedforward" network of pairs of neurons; each neuron, except the output neuron, and each external input, is associated to a "memory neuron". If $x_i(n)$ is the value of the output of neuron i and $x_j(n)$ the value of the output of the associated memory neuron j at time n, the output of the memory neuron at time n+1 is $x_j(n+1)= \alpha_i\, x_i(n) + (1-\alpha_i)\, x_j(n)$, $0<\alpha_i \leq 1$. If $\alpha_i=0$, the memory neurons introduce only delays, so that the network is a non-linear transversal filter. If $\alpha_i \neq 0$, the memory neurons are linear low-pass first order filters, and the network is actually a feedback network. A state output is associated to each memory neuron.
Figure A2.3a shows an example of such an architecture where neurons 3, 4, 7 and 8 are the memory neurons associated to the two inputs 1 and 2 and to the two neurons 5 and 6, respectively. The canonical form is shown in Figure A2.3b where x_3, x_4, x_7, x_8 are chosen as state variables.

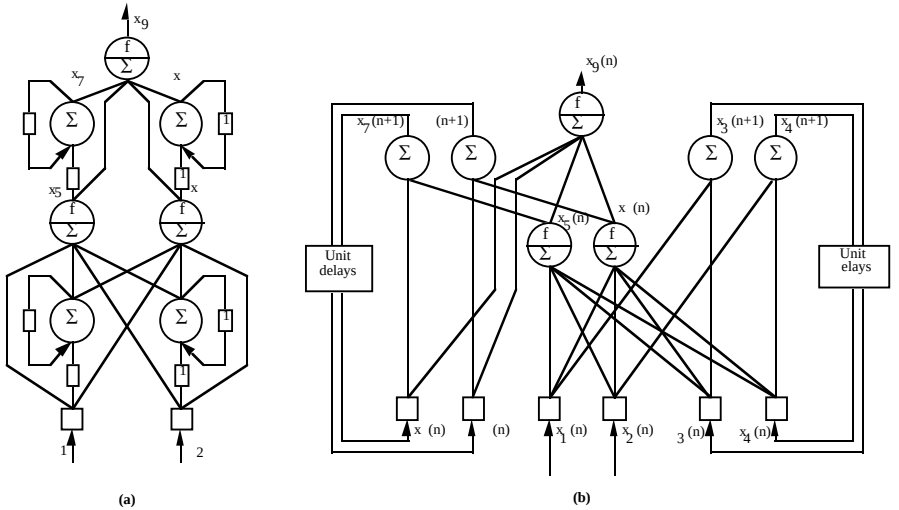

Figure A2.3:
a) Network architecture proposed by Poddar and Unnikrishnan.
b) Canonical form.

For process identification and control problems, the most general structure used by Narendra and Parthasarathy [Narendra and Parthasarathy 1991] is a model of the specific form: $y(n) = \psi_1[u(n-1), u(n-2), ...] + \psi_2[y(n-1), y(n-2), ...]$, where ψ_1 and ψ_2 are implemented by MLP networks with 20 neurons in the first hidden layer and 10 neurons in the second hidden layer.

APPENDIX 3

For simplicity, we present the training of the fully connected neural net of Figure 2: we denote the external inputs by z_1 to z_M , the feedback inputs by z_{M+1} to z_{M+N} , and the outputs of the neurons by z_{M+N+1} to $z_{M+N+\nu}$ (where ν is the number of neurons). The neurons are ordered in the following way: the p-th neuron receives the outputs of neurons indexed q<p (fully connected).

At time n, we have to consider the following cost function:

$$J(n) = \frac{1}{2} \sum_{m=N_t-N_c+1}^{N_t} (e^m)^2$$ where N_t is the number of blocks used to compute the N_c values e^m ($N \geq N_c$).

In this appendix, we present the contribution of block m ($1 \leq m \leq N_t$) to the gradient estimation. This contribution is computed from the external input vector, the desired value and the state input vector.

We denote the available values of the coefficients at time n by $\{c_{ij}\}$.

The canonical FF net of the m^{th} block, with coefficients $\{c_{ij}{}^m\}=\{c_{ij}\}$, computes the outputs $z_i^m=f_i(v_i^m)$ of all neurons and the state output vector $S_{out}{}^m(n)$ from the external input vector

$U^m(n) = [u(n-N_t+m),u(n-N_t+m-1),\ldots,u(n-N_t+m-M+1)] = [z_1^m, z_2^m, \ldots, z_M^m]$

and the state input vector $S_{in}^m(n) = [z_{M+1}^m, z_{M+2}^m, \ldots, z_{M+N}^m]$ as follow:

(i) For i = 1 to M (external inputs):
$z_i^m = u(n-N_t+m-i+1)$;

(ii) For i = M+1 to M+N (state inputs):
z_i^m is given by the chosen algorithm (table 1) ;

(iii) For i = M+N+1 to M+N+ν-1 (hidden neurons):
$z_i^m = f_i[v_i^m]$ with $v_i^m = \sum_{j \in P_i} c_{ij}^m z_j^m$;

(iv) For i = M+N+ν (linear output neuron):
$y^m = z_{M+N+\nu}^m = v_{M+N+\nu}^m = \sum_{j \in P_{M+N+\nu}} c_{M+N+\nu,j}^m z_j^m$.

Thus, the state output vector is
$S_{out}^m(n) \equiv [z_{M+N+\nu}^m, z_{M+N+\nu+1}^m, \ldots, z_{M+N+\nu+N-1}^m] = [y^m, z_{M+1}^m, \ldots, z_{M+N-1}^m]$.

and, if $N_t-N_c+1 \leq m \leq N_t$, we obtain from the desired value $d(n-N_t+m)$ and the output y^m:
$e^m = d(n-N_t+m) - y^m$.

In the following, we present two methods for the computation of the gradient of J(n): the forward computation and the backpropagation techniques.

1) Forward computation (Figure 4):

We consider the whole set of N_t blocks as a static network on which we perform the forward computation technique.

It is based on the following relation:

$$\frac{\partial J(n)}{\partial c_{ij}} = \frac{\partial}{\partial c_{ij}} \left(\frac{1}{2} \sum_{m=N_t-N_c+1}^{N_t} (e^m)^2 \right) = - \sum_{m=N_t-N_c+1}^{N_t} e^m \frac{\partial y^m}{\partial c_{ij}} .$$

The linear FC net of the m^{th} block computes, with coefficients $\{c_{ij}{}^m\}$ and $\{f'_i(v_i^m)\}$, the set of partial derivatives of the state output (including y^m) with respect to all coefficients c_{ij} :

$\frac{\partial S_{out}^m}{\partial c_{ij}}(n)$

For the $\nu(M+N)+(\nu-1)\nu/2$ coefficients c_{ij} (i>j):

(i) For p = 1 to M (external inputs):
$\frac{\partial z_p^m}{\partial c_{ij}} = 0$;

(ii) For p = M+1 to M+N (feedback inputs):
$\frac{\partial z_p^m}{\partial c_{ij}}$ is given by the chosen algorithm (table 2)

(iii) For p = M+N+1 to M+N+ν-1 (hidden neurons):
if p=i then $\frac{\partial z_i^m}{\partial c_{ij}} = f_i'[v_i^m] z_j^m$ otherwise $\frac{\partial z_p^m}{\partial c_{ij}} = f_p'[v_p^m] \sum_{h \in P_p} c_{ph} \frac{\partial z_h^m}{\partial c_{ij}}$;

(iv) For p = M+N+ν (linear output neuron):
$$\frac{\partial y^m}{\partial c_{ij}} = \frac{\partial z_{M+N+\nu}^m}{\partial c_{ij}} = \delta_{M+N+\nu,i} z_j^m + \sum_{h \in P_{M+N+\nu}} c_{M+N+\nu,h}^m \frac{\partial z_h^m}{\partial c_{ij}} .$$

Thus the partial derivatives of the state output are given by:
$$\frac{\partial S_{out}^m}{\partial c_{ij}}(n) \equiv \left[\frac{\partial z_{M+N+\nu}^m}{\partial c_{ij}} ; \frac{\partial z_{M+N+\nu+1}^m}{\partial c_{ij}} ; \ldots ; \frac{\partial z_{M+N+\nu+N-1}^m}{\partial c_{ij}} \right] = \left[\frac{\partial y^m}{\partial c_{ij}} ; \frac{\partial z_{M+1}^m}{\partial c_{ij}} ; \ldots ; \frac{\partial z_{M+N-1}^m}{\partial c_{ij}} \right]$$

Once all partial derivatives of the output values y^m are computed for the N_t blocks, the gradient of J(n) is obtained from:

$$\nabla J(n) = \left\langle \frac{\partial J(n)}{\partial c_{ij}} \right\rangle_{i>j} \text{ where } \frac{\partial J(n)}{\partial c_{ij}} = - \sum_{m=N_t-N_c+1}^{N_t} e^m \frac{\partial y^m}{\partial c_{ij}} .$$

If the steepest-descent method is used, the coefficient modifications are given by:

$$\Delta c_{ij}(n) = - \mu \frac{\partial J(n)}{\partial c_{ij}} = \mu \sum_{m=N_t-N_c+1}^{N_t} e^m \frac{\partial y^m}{\partial c_{ij}} = \sum_{m=N_t-N_c+1}^{N_t} \Delta c_{ij}^m(n).$$

2) Backpropagation (Figure 8):

Considering the effect of the coefficient c_{ij} only, one has:

$$dJ(n) = \sum_{m=1}^{N_t} \frac{\partial J(n)}{\partial c_{ij}^m} dc_{ij}^m \quad \text{with } dc_{ij}^m = dc_{ij} \ \forall\, m, \text{ thus } \quad \frac{dJ(n)}{dc_{ij}} = \sum_{m=1}^{N_t} \frac{\partial J(n)}{\partial c_{ij}^m} .$$

Then the gradient of J(n) can be written as:

$$\left(\frac{\partial J(n)}{\partial c_{ij}}\right)_{\substack{\forall p,q \neq i,j \\ c_{pq} \text{ constant}}} = \sum_{m=1}^{N_t} \left(\frac{\partial J(n)}{\partial c_{ij}^m}\right)_{\substack{\forall p,q \neq i,j \\ \forall m' \neq m \\ c_{pq}^{m'} \text{ constant}}}$$

where $\frac{\partial J(n)}{\partial c_{ij}^m} = \frac{\partial J(n)}{\partial v_i^m} \frac{\partial v_i^m}{\partial c_{ij}^m} = \frac{\partial J(n)}{\partial v_i^m} z_j^m$ $\quad$ for i=M+N+1 to M+N+ν, for j=1 to i-1

This means that standard backpropagation can be applied to the whole set of N_t blocks considered as a static network with replicated coefficients.

The linear BP net of the m^{th} block computes, with coefficients $\{c_{ij}^m\}$ and $\{f'_i(v_i^m)\}$, the set of partial derivatives of J(n) with respect to the potentials v_i^m of all neurons:

We define the following set of variables q_i^m:

(i) for i=M+N+ν+N-1 down to M+N+ν+1:

if $m=N_t$ then $q_i^m = 0$ otherwise $q_i^m = q_{i-N-\nu+1}^{m+1}$;

(ii) for i=M+N+ν (linear output neuron):

if $m=N_t$ then $q_i^m = e^m$ otherwise $q_i^m = e^m + q_{M+1}^{m+1}$; (note that $q_i^m = -\frac{\partial J(n)}{\partial v_i^m}$) ;

(iii) for i = M+N+ν−1 down to M+N+1 (hidden neurons):

$q_i^m = f_i'(v_i^m) \sum_{h \in R_i} c_{hi}^m q_h^m$; ($q_i^m = -\frac{\partial J(n)}{\partial v_i^m}$) ;

where R_i is the set of indices of the neurons to which the i-th neuron transmits its output

(iv) for i = M+N (last feedback input):

$q_i^m = \sum_{h \in R_i} c_{hi}^m q_h^m$;

(v) for i=M+N-1 down to M+1 (other feedback inputs):

$q_i^m = \sum_{h \in R_i} c_{hi}^m q_h^m + q_{i+N+\nu}^m$.

Note that computation by backpropagation assumes implicitly that the derivatives of the feedback inputs of the first block (m=1) with respect to the coefficients are equal to zero; this is in contrast to the forward computation of the gradient, where these values can be initialized arbitrarily.

Note also that with the forward computation technique, the number of partial derivatives to compute for each block is ν[νM+(ν-1)ν/2] whereas with the backpropagation method this number is ν.

Once all partial derivatives of J(n) with respect to the potentials v_i^m of all neurons are computed for the N_t blocks, the gradient of J(n) is obtained from:

$$\nabla J(n) = \left\{\frac{\partial J(n)}{\partial c_{ij}}\right\}_{i>j} \text{ where } \frac{\partial J(n)}{\partial c_{ij}} = \sum_{m=1}^{N_t} \frac{\partial J(n)}{\partial v_i^m} z_j^m .$$

If the steepest-descent method is used, the coefficient modifications are given by:

$$\Delta c_{ij}(n) = -\mu \frac{\partial J(n)}{\partial c_{ij}} = -\mu \sum_{m=1}^{N_t} \frac{\partial J(n)}{\partial v_i^m} z_j^m = \sum_{m=1}^{N_t} \Delta c_{ij}^m(n).$$

LITERATURE REFERENCES

Bellanger, M.G. 1987. *Adaptive Digital Filters and Signal Analysis*: Marcel Dekker.

Chen, S., Billings S.A. 1989. Representations of Non-Linear Systems: the NARMAX Model. *Int. J. Control* **49**, 1013-1032.

Chen, S., G.J. Gibson, C.F.N. Cowan and P.M. Grant. 1990. Adaptive Equalization of Finite Nonlinear Channels Using Multilayer Perceptrons. *Signal Processing* **20**, 107-119.

Dreyfus, G., O. Macchi, S. Marcos, L. Personnaz, P. Roussel-Ragot, D. Urbani, C. Vignat. 1992. Adaptive Training of Feedback Neural Networks for Non-linear Filtering and Control. In *Proceedings of the Second IEEE Conf. on Signal Processing.*

Elman, J. L. 1990. Finding Structure in Time. *Cognitive Science* **14**, 179-211.

Fallside, F. 1990. Analysis of Linear Predictive data as Speech and of ARMA Processes by a Class of Single-Layer Connectionnist Models. In *Neurocomputing: Algorithms, Architectures and Applications*, F. Fogelman-Soulié and J. Hérault, eds., 265-283.

Haykin, S. 1991. *Adaptive Filter Theory*: Prentice-Hall International Editions.

Hopfield, J.J. 1982. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proc . of the Natl. Acad. Sci. USA* 79, 2554-2558.

Hornik, K., M. Stinchcombe and H. White. 1989. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks* **2**, 359-366.

Jayant, N.S. and P. Noll. 1984. *Digital Coding of Waveforms. Principles and Applications to Speech and Video*. Signal Processing Series, A. Oppenheim, ed.: Prentice-Hall .

Jordan, M. I. 1985. *The Learning of Representations for Sequential Performace*. Doctoral Dissertation, University of California, San Diego.

Jordan, M. I. 1989. Serial Order: A Parallel, Distributed Processing Approach. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 531-546: Erlbaum.

Lapedes, A., and R. Farber. How Neural Nets Work. 1988. In *Neural Information Processing Systems*, D. Z. Anderson, ed., 442-456.

McCannon, T.E., N.C. Gallagher, D.Minoo-Hamedani and G.L. Wise. 1982. On the design of nonlinear discrete-time predictors. *IEEE Trans. on Information Theory* **28**, 366-371.

Narendra, K. S., and K. Parthasarathy. 1990. Identification and Control of Dynamical Systems Using Neural Networks. *IEEE Transactions on Neural Networks*, **1**, 4-27.

Narendra, K. S., and K. Parthasarathy. 1991. Gradient Methods for the Optimization of Dynamical Systems Containing Neural Networks. *IEEE Trans. on Neural Networks* **2**, 252-262.

Nicolau, E. and D. Zaharia. 1989 . Adaptive arrays. In *Studies in Electrical and Electronic Engineering 35*: Elsevier.

Pearlmutter B. 1989. Learning State Space Trajectories in recurrent Neural Networks. *Neural Computation* **1**, 263-269.

Personnaz L., I. Guyon and G. Dreyfus. 1986. Collective Computational Properties of Neural Networks: New Learning Mechanisms. *Phys. Rev. A* **34**, 4217-4228.

Picinbono, B. 1988. Adaptive methods in temporal processing. In *Underwater Acoustic Data Processing*, Y.T. Chan, ed., 313-327. Kluwer academic Publishers.

Pineda F. J. 1989. Recurrent Backpropagation and the Dynamical Approach to Adaptive Neural Computation. *Neural Comp.* **1**, 161-172.

Pineda, F. 1987. Generalization of Backpropagation to Recurrent Neural Networks. *Phys. Rev. Lett.* **59**, 2229-2232.

Poddar, P., and K.P. Unnikrishnan. 1991. Non-Linear Prediction of Speech Signals using Memory Neuron Networks. In *Neural Networks for Signal Processing, Proceedings of the 1991 IEEE Workshop*, B. H. Juang, S. Y. Kung, and C. A. Kamm, eds., 395-404.

Press, W.H., B.P. Flannery, S.A. Teukolsky, W.T. Vetterling. 1986. *Numerical Recipes.* Cambridge University Press.

Proakis, J.G. 1983. *Digital communications*: Mc Graw Hill.

Robinson, A. J. and F. Fallside. 1989. A Dynamic Connectionnist Model for Phoneme Recognition. In *Neural Networks from Models to Applications*, L. Personnaz and G. Dreyfus, eds., 541-550: Paris, IDSET.

Rumelhart, D., G. Hinton, R. Williams. 1986. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing,* D. Rumelhart and J. McClelland, eds. MIT Press.

Shynk, J.J. 1989. Adaptive IIR Filtering. *IEEE ASSP Magazine.* April, 4-21.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. 1989. Phoneme Recognition Using Time-Delay Neural Networks. *IEEE Trans. on Acoustics, Speech, and Signal Processing* **37**, 328-339.

Weigend, A. S., B.A. Huberman and D.E. Rumelhart. 1990. Predicting the Future: a Connectionnist Approach. *International Journal of Neural Systems* **1**, 193-209.

Widrow, B. and S.D. Stearns. 1985. *Adaptive Signal Processing*: Prentice-Hall.

Williams, R.J. and D. Zipser. 1989a. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Comp.* **1**, 270-280.

Williams, R.J. and D. Zipser. 1989b. Experimental Analysis of the Real-Time Recurrent Learning Algorithm. *Connection Science* **1**, 87-111.

Williams, R.J. and J. Peng. 1990. An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories. *Neural Comp.* **2**, 490-501.