A Signal Processing Framework Based on Dynamic Neural Networks with Application to Problems in Adaptation, Filtering, and Classification

LEE A. FELDKAMP, SENIOR MEMBER, IEEE, AND GINTARAS V. PUSKORIUS, MEMBER, IEEE

Invited Paper

We present in this paper a coherent neural network-based framework for solving a variety of difficult signal processing problems. The framework relies on the assertion that time-lagged recurrent networks possess the necessary representational capabilities to act as universal approximators of nonlinear dynamical systems. This property applies to modeling problems posed as system identification, time-series prediction, nonlinear filtering, adaptive filtering, and temporal pattern classification. We address the development of models of nonlinear dynamical systems, in the form of time-lagged recurrent neural networks, which can be used without further training (i.e., as fixed-weight networks). We employ a weight update procedure based on the extended Kalman filter (EKF); as a solution to the recency effect, which is the tendency for a network to forget earlier learning as it processes new examples, we have developed a technique called multistream training.

We demonstrate our training framework by applying it to four problems. First, we show that a single time-lagged recurrent neural network can be trained not only to produce excellent one-time-step predictions for two different time series, but also to be robust to severe errors in the provided input sequence. The second problem involves the modeling of a complex system containing significant process noise, which was shown in [1] to lead to unstable trained models. We illustrate how multistream training may be used to enhance the stability of such models.

The remaining two problems are drawn from real-world automotive applications. The first of these involves input–output modeling of the dynamic behavior of a catalyst-sensor system which is exposed to an operating engine's exhaust stream. Finally we consider real-time and continuous detection of engine misfire, which is cast as a dynamic pattern classification problem.

Keywords*— *Automotive diagnostics, backpropagation through time, Kalman filtering, multistream training, recurrent multilayer perceptrons, recurrent networks, stability, system identification, time series prediction.

Manuscript received November 1, 1997; revised April 17, 1998.

The authors are with the Ford Research Laboratory, Ford Motor Company, Dearborn, MI 48121-2053 USA.

Publisher Item Identifier S 0018-9219(98)07862-1.

I. INTRODUCTION

We consider in this paper a practical and quite general framework with which we have approached many different types of problems that involve the temporal processing of signals. Some of these problems have been cast into a more or less standard format, i.e., a single signal sampled in time, upon which we are expected to perform prediction, filtering, or estimation. More commonly, however, the problems we have encountered have had more of a mixed character. In particular, such problems usually involve a primary signal and one or more additional signals which, taken together, provide context. The problem statement might then involve prediction, estimation (particularly in the form of virtual sensors) or classification. Furthermore, we usually find that the systems involved cannot really be regarded as stationary; on the other hand, they are not subject to unlimited variation. Perhaps the best description is a progression among several ill-defined modes of operation. The systems we deal with are usually driven rather than autonomous, though we seldom have access to the underlying drivers.

Our philosophy has been to take a fairly general input–output point of view, in which we assess the available input information sequence and seek to transform it into the required output sequence. Implicit in this view is that we can, in fact, assemble a desired output sequence. (This may be contrasted with problems such as certain types of control, in which desired system outputs are not necessarily available at every moment, and in which the desired output of a neural controller is not provided and must be constructed. The methods described here have been used in such applications as well.)

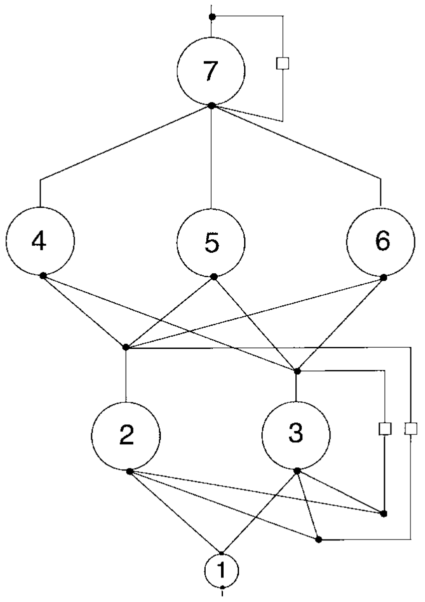
Our technical approach has been to develop a methodology for effective training of time-lagged recurrent networks, usually in the form of recurrent multilayer perceptrons

(RMLP). The latter are a natural synthesis of feedforward multilayer perceptrons and single-layer fully recurrent networks. For an introduction to neural networks, the reader may wish to consult a textbook such as [2] or [3].

From a structural point of view, RMLP's are appealing by virtue of subsuming many traditional signal processing structures, including tapped delay lines, finite impulse response (FIR) filters, infinite impulse response (IIR) filters, and gamma networks, while retaining the universal approximation property of feedforward networks. Because they have state variables, RMLP's can represent dynamic systems with strongly hidden state [4]; indeed, Lo [5] suggests that RMLP's are universal approximators for dynamic systems, just as feedforward MLP's are universal approximators for static mappings (see, for example, [6]). Further, the work of Cotter and Conwell [7] suggests that time-lagged fixed-weight recurrent networks can produce behavior that would usually be called adaptive. In [9] and [10] we presented an example of such behavior. Though the referenced work was purely abstract, it was motivated by a contemporaneous practical application involving a system that exhibits nonstationarity over a bounded range.

Recurrent networks have often been regarded as difficult to train. Although we contend that an effective procedure can be carried out relatively routinely, the difficulties presented in training are not mere illusions. In addition to compounding the usual pitfalls encountered in training feedforward networks, including poor local minima, recurrent networks enhance the difficulty of training by the recency effect. Simply stated, a tendency always exists for recent weight updates to cause a network to forget what it has learned in the past. Of course, this tendency exists also in the training of feedforward networks, but in that case easy and effective countermeasures exist, viz., scrambling the order of presentation of input–output pairs or employing batch learning (in which an update may be based on all examples). Such methods are cumbersome to employ in training recurrent networks because the temporal order of the data sequences must be respected. Although recurrent networks can be trained with first- or second-order batch weight update procedures, we prefer to utilize sequential training methods because of frequent weight updates and the benefits of their stochastic characteristics.

In the remainder of this paper, we have organized our presentation as follows. Section II presents briefly the RMLP architecture and notes how various signal processing structures can be formed explicitly. Section III begins a discussion of the training method by describing the calculation of derivatives (gradients). Section IV describes how these derivatives are used in a second-order weight update procedure, based on the extended Kalman filter. Section IV also describes our approach to mitigating the recency effect, which we term multistream training (or simply multistreaming). Here we also relate an enabling side benefit of multistreaming: the ability to coerce desired secondary network behavior at the same time the network is being trained on its primary objective. In Section V we discuss two synthetic examples. First, we describe a



i	j	$c_{i,j}$	$d_{i,j}$
2	1	0	0
	2	1	0
	3	2	1
	4	3	1
3	1	0	0
	2	1	0
	3	2	1
	4	3	1
4	1	0	0
	2	2	0
	3	3	0
5	1	0	0
	2	2	0
	3	3	0
6	1	0	0
	2	2	0
	3	3	0
7	1	0	0
	2	4	0
	3	5	0
	4	6	0
	5	7	1

Fig. 1. Schematic illustration of an RMLP, denoted as 1-2R-3-1R. The first hidden layer of two nodes is fully recurrent, the second hidden layer (three nodes) is feedforward and the output node is recurrent. The small boxes denote unit time delays. The table contains elements of the connection and delay arrays. Entries for index i are absent for the bias (treated as node 0) and the network input (node 1), since neither receives input from any other node. Bias connections are present, as indicated in the connection table, but they are not drawn. Note that recurrent connections have unit delays. The elements of the input and output arrays are: $I_1 = 1$ and $O_1 = 7$.

problem related to the multiple system modeling of [9] and [10], with the added requirement of dealing with dropouts in a faulty data sequence. Then we illustrate the modeling of a complex system, presented in [1], together with a practical approach to imposing stability on the neural model. Section VI is devoted to problems taken from real-world systems. The first of these is a modeling/estimation problem; beyond merely complicated behavior, this application requires that one deal with delays that vary substantially over the operating range of the system. Then we discuss a classification problem in which faults must be detected on the basis of an apparently hopelessly noisy input sequence. Finally, in Section VII we summarize and make some concluding remarks.

II. Network Architecture and Execution

An RMLP consists of one or more layers of computing nodes, just as in a standard feedforward network or MLP. A simple RMLP architecture is illustrated in Fig. 1. In the basic RMLP form we either have full recurrence within a layer, which means that every node is connected through unit time delays to every node in the same layer, or else no recurrence for that layer. (A convenient feature of the RMLP architecture is that it reduces to a simple MLP in the absence of recurrence.) Recurrence can also be present from network outputs to network inputs. It is also possible to have a sparse connection pattern. Indeed, it is the use of sparse connections together with flexible assignment of node activation functions that permit an RMLP to be structured to act as traditional linear filters [e.g., FIR, IIR], gamma networks, time-delay networks, Elman and Jordan networks, and other useful computational forms. This means, of course, that a basic RMLP can, in principle, be trained to subsume any structured network which it contains, although success in training cannot be guaranteed.

It is possible to restructure an RMLP so that it is expressed as a pure feedforward network with external (output–input) recurrent connections. An awkward aspect of this transformation is that it is necessary to replicate some of the nodes. Though convenient for some purposes, we feel that such a restructuring contributes little if anything to the training process.

For compactness of presentation, we shall express the operation of an RMLP by treating it as a special case of an ordered network [11]. The forward equations for an ordered network with n_in inputs and n_out outputs may be expressed very compactly, using a pseudocode format similar to that in [12].

Let the network consist of n_nodes nodes, including n_in nodes which serve as receptors for the external inputs, but not including the bias input, which we denote formally as node 0. The latter is set to the constant 1.0. The array $\boldsymbol{I}$ contains a list of the input nodes, e.g., I_j is the number of the node that corresponds to the $\mathtt{j}$th input, in_j. Similarly, a list of the nodes that correspond to network outputs out_p is contained in the array $\boldsymbol{O}$. We allow for the possibility of network outputs and targets to be advanced or delayed with respect to node outputs by assigning a phase τ_p to each output. In most cases these phases are zero. Node i receives input from $\mathtt{n\_con(i)}$ other nodes and has activation function $f_i(\cdot)$; $\mathtt{n\_con(i)}$ is zero if node $\mathtt{i}$ is among the nodes listed in the array $\boldsymbol{I}$. The array $\boldsymbol{c}$ specifies connections between nodes; $\mathrm{c}_{i,j}$ is the node number for the $\mathtt{j}$th input for node $\mathtt{i}$. Inputs to a given node may originate at the current or past time steps, according to delays contained in the array $\boldsymbol{d}$, and through weights for time step t contained in the array $\boldsymbol{W}(t)$. Summarizing, the $\mathtt{j}$th input to node $\mathtt{i}$ at time step t is the output of node $\mathrm{c}_{i,j}$ at time $t-d_{i,j}$ and is connected through weight $W_{i,j}(t)$.

Prior to beginning network operation, all appropriate memory is initialized to zero. At the beginning of each time step, we execute the following buffer operations on weights and node outputs (in practical implementation, a circular buffer and pointer arithmetic may be employed). Here dmax is the largest value of delay represented in the array $\boldsymbol{d}$, and h is the truncation depth of the backpropagation through time gradient calculation that is described in the following section.

```
for i = 1 to n_nodes{
for i_t = t - h - dmax to t - 1 {
        $\boldsymbol{W}(i_t) = \boldsymbol{W}(i_t + 1)$        (1)
        $y_i(i_t) = y_i(i_t + 1)$        (2)
}}
```

Then the actual network execution is expressed as

```
for i = 1 to n_in {
        $y_{\mathrm{I}_i}(t) = \mathrm{in}_i(t)$        (3)
}
for i = 1 to n_nodes {
if n_con(i) > 0 {
        $y_i(t) = f_i\left(\sum_{j=1}^{\mathrm{n\_con(i)}} W_{i,j}(t) y_{\mathrm{c}_{i,j}}(t - d_{i,j})\right)$        (4)
}
}
for p = 1 to n_out {
        $\mathrm{out}_p(t + \tau_p) = y_{\mathrm{O}_p}(t)$        (5)
}.
```

This description of an ordered network does not explicitly involve the concept of layers; the layered structure is imposed implicitly by the connection pattern. Pure delays can be described directly, so that tapped delay lines on either external or recurrent inputs are conveniently represented.

III. Gradient Calculation

After the forward propagation at time step t, we compute gradients in preparation for the weight update step. In the past we have made extensive use of forward methods of derivative calculation [13], [14]. Some time ago, however, we replaced the forward method with a form of truncated backpropagation through time (BPTT(h)) [11], [15]. With the truncation depth h suitably chosen, this method produces derivatives that closely approximate those of the forward method with greatly reduced complexity and computational effort.

We describe here the mechanics of the particular form of BPTT(h) we have most commonly employed. We use the Werbos notation in which $\mathrm{F}\underline{}^{q}x$ denotes an ordered derivative of some quantity q with respect to x. In this form of BPTT, $\mathrm{F}\underline{}^{q}x$ denotes the ordered derivative of network output node p with respect to x.

To derive the backpropagation equations, the forward propagation equations are considered in reverse order. From each we derive one or more backpropagation expressions, according to the principle that if $a = g(b, c)$, then $\mathrm{F}\underline{}^{q}b \mathrel{+}= (\partial g/\partial b)\mathrm{F}\underline{}^{q}a$ and $\mathrm{F}\underline{}^{q}c \mathrel{+}= (\partial g/\partial c)\mathrm{F}\underline{}^{q}a$. The C-language notation "$+=$" indicates that the quantity on the right-hand side is to be added to the previously computed value of the left-hand side. In this way, the appropriate derivatives are distributed from a given node to all nodes and weights that feed it in the forward direction, with due allowance for any delays that might be present in each connection. The simplicity of the formulation reduces the need for visualizations such as unfolding in time or signal-flow graphs.

```
for p = 1 to n_out {
for i = 1to n_nodes {
for i_t = t to t − h − 1 {
                $F\underline{\ p}\, y_i(i_t) = 0$    (6)
}} /* end i and i_t loops */
                $F\underline{\ p}\, y_{O_p}(t) = 1$    (7)
        $\xi_p(t) = \mathrm{tgt}_p(t+\tau_p) - \mathrm{out}_p(t+\tau_p)$    (8)
for i_h = 0 to h {
                $i_1 = t - i_h$    (9)
for i = n_nodes to 1 {
if n_con(i) > 0 {
for k = n_con(i) to 1 {
                $j = c_{i,k}$    (10)
                $i_2 = i_1 - d_{i,k}$    (11)
    $F\underline{\ p}\, y_j(i_2) \mathrel{+}= \gamma^{d_{i,k}} F\underline{\ p}\, y_i(i_1) W_{i,k}(i_1) f'(y_i(i_1))$    (12)
        $F\underline{\ p}\, W_{i,j} \mathrel{+}= y_j(i_2) F\underline{\ p}\, y_i(i_1) f'(y_i(i_1))$    (13)
} /* end k loop */
}
} /* end i loop */
} /* end i_h loop */
} /* end p loop */
```

Here (6) serves to initialize the derivative array, while (7) expresses the fact that $(\partial y_{O_p}(t))/(\partial y_{O_p}(t)) = 1$. The error $\xi_p(t)$ computed in (8) is used in the weight update described in the next section, where the desired value of network output $\mathrm{out}_p(t+\tau_p) = y_{O_p}(t)$ is denoted as $\mathrm{tgt}_p(t+\tau_p)$. The actual backpropagation occurs in expressions (12) and (13), which derive directly from the forward propagation expression (4). We have included a discount factor γ in expression (12), though it is often set merely to its nominal value of unity.

IV. Extended Kalman Filter (EKF) Multistream Training

A. The Kalman Recursion

We have made extensive use of training that employs weight updates based on the EKF method first proposed by Singhal and Wu [16]. (For background material on the Kalman filter, see [17] and [18].) In most of our work, we have made use of a decoupled version of the EKF method [14], [19] which we denote as DEKF. Decoupling was crucial for early practical use of the method, when speed and memory capabilities of workstations and personal computers were severely limited. At the present time, many problems are small enough to be handled with what we have termed global EKF, or GEKF. In many cases, the added coupling brings benefits in terms of quality of solution and overall training time. However, the increased time required for each GEKF update is a potential disadvantage in real-time applications.

For generality, we present the decoupled Kalman recursion; GEKF is recovered in the limit of a single weight group ($g = 1$). The weights in $\boldsymbol{W}$ are organized into g mutually exclusive weight groups; a convenient and effective choice has been to group together those weights that feed each node. Whatever the chosen grouping, the weights of group i are denoted by $\boldsymbol{w}_i$. The corresponding derivatives $F\underline{\ p}\,\boldsymbol{w}_i$ are placed in n_out columns of $\boldsymbol{H}_i$.

To minimize a cost function $E = \Sigma_t\ \frac{1}{2}\ \boldsymbol{\xi}(t)^T \boldsymbol{S}(t)\boldsymbol{\xi}(t)$, where $\boldsymbol{S}(t)$ is a nonnegative definite weighting matrix and $\boldsymbol{\xi}(t)$ is the vector of errors, at time step t, the recursion equations are as follows [12]:

$$\boldsymbol{A}^*(t) = \left[\frac{1}{\eta(t)}\boldsymbol{I} + \sum_{j=1}^{g} \boldsymbol{H}_j^*(t)^T \boldsymbol{P}_j(t) \boldsymbol{H}_j^*(t)\right]^{-1} \quad (14)$$

$$\boldsymbol{K}_i^*(t) = \boldsymbol{P}_i(t)\boldsymbol{H}_i^*(t)\boldsymbol{A}^*(t) \quad (15)$$

$$\boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t) + \boldsymbol{K}_i^*(t)\boldsymbol{\xi}^*(t) \quad (16)$$

$$\boldsymbol{P}_i(t+1) = \boldsymbol{P}_i(t) - \boldsymbol{K}_i^*(t)\boldsymbol{H}_i^*(t)^T\boldsymbol{P}_i(t) + \boldsymbol{Q}_i(t). \quad (17)$$

In these equations, the weighting matrix $\boldsymbol{S}(t)$ is distributed into both the derivative matrices and the error vector: $\boldsymbol{H}_i^*(t) = \boldsymbol{H}_i(t)\boldsymbol{S}(t)^{1/2}$ and $\boldsymbol{\xi}^*(t) = \boldsymbol{S}(t)^{1/2}\boldsymbol{\xi}(t)$. The matrices $\boldsymbol{H}_i^*(t)$ thus contain the scaled derivatives of network outputs with respect to the ith group of weights; the concatenation of these matrices forms a global scaled derivative matrix $\boldsymbol{H}^*(t)$. A common global scaling matrix $\boldsymbol{A}^*(t)$ is computed with contributions from all g weight groups, through the scaled derivative matrices $\boldsymbol{H}_j^*(t)$ and from all of the decoupled approximate error covariance matrices $\boldsymbol{P}_j(t)$. A user-specified learning rate $\eta(t)$ appears in this common matrix. For each weight group i, a Kalman gain matrix $\boldsymbol{K}_i^*(t)$ is computed and is then used in updating the values of the group's weight vector $\boldsymbol{w}_i(t)$ and in updating the group's approximate error covariance matrix $\boldsymbol{P}_i(t)$. Each approximate error covariance update is augmented with the addition of a scaled identity matrix $\boldsymbol{Q}_i(t)$ that represents the effects of artificial process noise.

In practice, the EKF recursion is typically initialized by setting the approximate error covariance matrices to scaled identity matrices, with a scaling factor of 100 for nonlinear nodes and 1000 for linear nodes. At the beginning of training, we generally set the learning rate low (the actual value depends on characteristics of the problem, but $\eta = 0.1$ is a typical value), and start with a relatively large amount of process noise, e.g., $\boldsymbol{Q}_i(0) = 10^{-2}\eta\boldsymbol{I}$. We have previously demonstrated that the artificial process noise extension accelerates training, helps to avoid poor local minima during training, and assists the training procedure in maintaining the necessary property of nonnegative definiteness for the approximate error covariance matrices [19]. As training progresses, we generally decrease the amount of process noise to a limiting value of approximately $\boldsymbol{Q}_i(t) = 10^{-6}\boldsymbol{I}$ and increase the learning rate to a limiting value no greater than unity. In addition, we have also found that occasional reinitializations of the error covariance matrices, along with resetting of initial values for the learning rate and process noise terms, may benefit the training process. Finally, one should note that initial choices for the learning rate and error covariance matrices are not independent: a multiplicative increase in the scaling factor for the approximate error covariance matrices can be cancelled by reducing the initial learning rate by the inverse of the scaling factor (the relative scalings of the learning rate and error covariance matrices affect the choice of the artificial process noise term as well).

We wish to emphasize here that the decoupled EKF recursion given by (14)–(17) only performs updates to network weights, and not to the states of the dynamical network model. On the other hand, Matthews [20] and Williams [21] have described parallel EKF formulations for recurrent networks in which both network states and weights are updated during training. However, these parallel formulations present certain difficulties. First, the parallel techniques require that gradients be computed by forward methods, and it is not obvious how computationally efficient methods such as BPTT(h) can be used within the parallel formalism. Similarly, we are primarily interested in training recurrent networks to deploy as fixed-weight systems without having to incur the computational cost of performing the Kalman recursion for state estimation during network use. Finally we have found, as shown below, that properly trained fixed-weight recurrent networks appear to be capable of performing many filtering, estimation, and prediction tasks that are often thought to be suitable candidates for traditional extended Kalman filtering, and it is not obvious that additional filtering of model states would be beneficial.

B. Multistream Training

Consider the standard recurrent network training problem of training on a sequence of input–output pairs. If the sequence is in some sense homogeneous, then one or more linear passes through the data may well produce good results. However in many training problems, especially those in which exogenous inputs are present, the data sequence is heterogeneous. For example, regions of rapid variation of inputs and outputs may be followed by regions of slow change. Or a sequence of outputs that centers about one level may be followed by one that centers about a different level. For any of these cases, in a straightforward training process the tendency always exists for the network weights to be adapted unduly in favor of the currently presented training data. This recency effect is analogous to the difficulty that may arise in training feedforward networks because of the order in which the training data are presented.

In this latter case, an effective solution is to scramble the order of presentation; another is to use a batch update algorithm. For recurrent networks, the direct analog of scrambling the presentation order is to present randomly selected subsequences, making an update only for the last input–output pair of the subsequence (when the network would be expected to be independent of its initialization at the beginning of the sequence). A full batch update involves running the network through the entire data set, computing the required derivatives that correspond to each input–output pair, and making an update based on the entire set of errors.

The multistream procedure largely circumvents the recency effect by combining features of both scrambling and batch updates. Like full batch methods, multistream training [22] is based on the principle that each weight update should attempt to satisfy simultaneously the demands from multiple input–output pairs. It retains, however, the useful stochastic aspects of sequential updating and requires much less computation time between updates. We now describe the mechanics of multistream training.

In a typical training problem we deal with one or more files, each of which contains a sequence of data. Breaking the overall data set into multiple files is typical in practical problems, where the data may be acquired in different sessions, for distinct modes of system operation, or under different operating conditions.

In each cycle of training, we choose a specified number N_s of randomly selected starting points in a chosen set of files. Each such starting point is the beginning of a stream. The multistream procedure consists of progressing in sequence through each stream, carrying out weight updates according to the set of current points. Copies of recurrent node outputs must be maintained separately for each stream. Derivatives are also computed separately for each stream, generally by truncated BPTT(h) as discussed above. Because we generally have no prior information with which to initialize the recurrent network, we typically set all state nodes to values of zero at the start of each stream. Accordingly, the network is executed but updates are suspended for a specified number N_p of time steps, called the priming length, at the beginning of each stream. Updates are performed until a specified number N_t of time steps, called the trajectory length, have been processed. Hence $N_t - N_p$ updates are performed in each training cycle.

If we take $N_s = 1$ and $N_t - N_p = 1$, we recover the order-scrambling procedure described above; N_t may be identified with the subsequence length. On the other hand, we recover the batch procedure if we take N_s equal to the number of time steps for which updates are to be performed, assemble streams systematically to end at the chosen N_s steps, and again take $N_t - N_p = 1$.

Generally speaking, apart from the computational overhead involved (see below), we find that performance tends to improve as the number of streams is increased. Various strategies are possible for file selection. If the number of files is small, it is convenient to choose N_s equal to a multiple of the number of files and to select each file the same number of times. If the number of files is too large to make this practical, then we tend to select files randomly. In this case, each set of $N_t - N_p$ updates is based on only a subset of the files, so it seems reasonable not to make the trajectory length N_t too large.

An important consideration is how to carry out the EKF update procedure. If first-order gradient updates were being used, we would simply average the updates that would have

$$\mathbf{H} = \left(\begin{array}{c|c|c|c|c|c} H_{\substack{p=1\\s=1}} & H_{\substack{p=2\\s=1}} & H_{\substack{p=1\\s=2}} & H_{\substack{p=2\\s=2}} & H_{\substack{p=1\\s=3}} & H_{\substack{p=2\\s=3}} \end{array}\right)$$

$$\xi = \left(\xi_{\substack{p=1\\s=1}}\ \xi_{\substack{p=2\\s=1}}\ \xi_{\substack{p=1\\s=2}}\ \xi_{\substack{p=2\\s=2}}\ \xi_{\substack{p=1\\s=3}}\ \xi_{\substack{p=2\\s=3}}\right)^T$$

Fig. 2. Illustration of the augmentation of the derivative matrix and error vector for the case of two outputs and three streams.

been performed had the streams been treated separately. In the case of EKF training, however, averaging separate updates is incorrect. Instead, we treat this problem as that of training a single shared-weight network with $N_s \times \texttt{n\_out}$ outputs. From the standpoint of the EKF method, we are simply training a multiple output network in which the number of original outputs is multiplied by the number of streams. The nature of the Kalman recursion is then to produce weight updates which are not a simple average of the weight updates that would be computed separately for each output, as is the case for a simple gradient descent weight update.

In single-stream EKF training we place derivatives of network outputs with respect to network weights in the matrix $\boldsymbol{H}$ constructed from n_out column vectors, each of dimension equal to the number of trainable weights, N_w. In multistream training, the number of columns is correspondingly increased to $N_s \times \texttt{n\_out}$. Similarly, the vector of errors $\boldsymbol{\xi}$ has $N_s \times \texttt{n\_out}$ elements. Apart from these augmentations of $\boldsymbol{H}$ and $\boldsymbol{\xi}$, illustrated schematically in Fig. 2, the form of the Kalman recursion is unchanged.

Let us consider the computational implications of the multistream method. The sizes of the approximate error covariance matrices $\boldsymbol{P}_i$ and the weight vectors $\boldsymbol{w}_i$ are independent of the chosen number of streams. The number of columns of the derivative matrices $\boldsymbol{H}_i^*$, as well as of the Kalman gain matrices $\boldsymbol{K}_i^*$, increases from n_out to $N_s \times \texttt{n\_out}$, but the computation required to obtain $\boldsymbol{H}_i^*$ and to compute updates to $\boldsymbol{P}_i$ is the same as for N_s separate updates. The major additional computational burden is the inversion required to obtain the $\boldsymbol{A}^*$ matrix, whose dimension is N_s times larger. Even this tends to be small compared to the cost associated with propagating the $\boldsymbol{P}_i$ matrices, as long as $N_s \times \texttt{n\_out}$ is smaller than the number of network weights (GEKF) or the maximum number of weights in a group (DEKF).

If the number of streams chosen is so large as to make the inversion of $\boldsymbol{A}^*$ impractical, the inversion may be avoided by treating the multiple network outputs with a scalar cost function as described in [23]. The efficacy of multistream training performed in this fashion remains to be explored thoroughly, as the presumed advantage of pseudoinverse-like updates, as described below, may be lost.

C. Some Insight into the Multistream Technique

A simple means of motivating how multiple training instances can be used simultaneously for a single weight update via the EKF procedure is to consider the training of a single linear node. In this case, the application of EKF training is equivalent to that of recursive least squares (RLS). (A recent discussion of the relationship between RLS and the KF may be found in [24].) Assume that a training data set is represented by m unique training patterns. The ith training pattern is represented by a d-dimensional input vector $\boldsymbol{x}(i)$, where we assume that all input vectors include a constant bias component of value equal to one and a one-dimensional output target $y(i)$. The simple linear model for this system is given by

$$\hat{y}(i) = \boldsymbol{x}(i)^T \boldsymbol{w}_f \tag{18}$$

where $\boldsymbol{w}_f$ is the single node's d-dimensional weight vector. The weight vector $\boldsymbol{w}_f$ can be found by applying m iterations of the RLS procedure as follows:

$$a(i) = \left[1 + \boldsymbol{x}(i)^T \boldsymbol{P}(i)\boldsymbol{x}(i)\right]^{-1} \tag{19}$$

$$\boldsymbol{k}(i) = \boldsymbol{P}(i)\boldsymbol{x}(i)a(i) \tag{20}$$

$$\boldsymbol{w}(i+1) = \boldsymbol{w}(i) + \boldsymbol{k}(i)(y(i) - \hat{y}(i)) \tag{21}$$

$$\boldsymbol{P}(i+1) = \boldsymbol{P}(i) - \boldsymbol{k}(i)\boldsymbol{x}(i)^T \boldsymbol{P}(i) \tag{22}$$

where the diagonal elements of $\boldsymbol{P}$ are initialized to large positive values, here $\boldsymbol{P}(0) = 1000\boldsymbol{I}$, and $\boldsymbol{w}(0)$ to a vector of small random values. Also, $\boldsymbol{w}_f = \boldsymbol{w}(m)$ after a single presentation of all training data (i.e., after a single epoch).

We recover a batch, LS solution to this single-node training problem via an extreme application of the multistream concept, where we associate m unique streams with each of the m training instances. In this case, we arrange the input vectors into a matrix $\boldsymbol{X}$ of size $d \times m$, where each column corresponds to a unique training pattern. Similarly, we arrange the target values into a single m-dimensional vector $\boldsymbol{y}$, where each element of $\boldsymbol{y}$ is properly aligned with its corresponding feature vector in $\boldsymbol{X}$. As before, we select the initial weight vector $\boldsymbol{w}(0)$ to consist of randomly chosen values, and we select $\boldsymbol{P}(0) = 1000\boldsymbol{I}$. Given the choice of initial weight vector, we can compute the network output for each training pattern and arrange all the results using the matrix notation

$$\hat{\boldsymbol{y}}(0) = \boldsymbol{X}^T \boldsymbol{w}(0). \tag{23}$$

A single weight update step of the Kalman filter recursion applied to this m-dimensional output problem at the beginning of training can be written as

$$\boldsymbol{A}(0) = \left[\boldsymbol{I} + \boldsymbol{X}^T \boldsymbol{P}(0)\boldsymbol{X}\right]^{-1} \tag{24}$$

$$\boldsymbol{K}(0) = \boldsymbol{P}(0)\boldsymbol{X}\boldsymbol{A}(0) \tag{25}$$

$$\boldsymbol{w}(1) = \boldsymbol{w}(0) + \boldsymbol{K}(0)(\boldsymbol{y} - \hat{\boldsymbol{y}}(0)) \tag{26}$$

where we have chosen not to include the error covariance update here for reasons that will soon become obvious. At the beginning of training, we recognize that $\boldsymbol{P}(0)$ is large, and we assume that the training data set is scaled so that $\boldsymbol{X}^T\boldsymbol{P}(0)\boldsymbol{X} \gg \boldsymbol{I}$. This allows $\boldsymbol{A}(0)$ to be approximated by

$$\boldsymbol{A}(0) \approx \frac{1}{1000}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1} \tag{27}$$

where we have taken advantage of the diagonal nature of $\boldsymbol{P}(0)$. Given this approximation, we can write the Kalman gain matrix as

$$\boldsymbol{K}(0) = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}. \tag{28}$$

We now substitute (23) and (28) into (26) to derive the weight vector after one time step of this m-stream Kalman filter procedure

$$\boldsymbol{w}(1) = \boldsymbol{w}(0) + \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\left(\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{w}(0)\right) \tag{29}$$

$$= \boldsymbol{w}(0) - \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{w}(0) + \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{y}. \tag{30}$$

Applying the matrix equality $\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T = \boldsymbol{I}$ yields the pseudoinverse solution to this training problem

$$\boldsymbol{w}_f = \boldsymbol{w}(1) = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{y} \tag{31}$$

since $\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1} = (\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{X}$.

Thus one step of the multistream Kalman recursion recovers very closely the LS solution. If m is too large to make the inversion operation practical, we could instead divide the problem into subsets and perform the procedure sequentially for each subset, arriving eventually at nearly the same result (in this case, however, the covariance update needs to be performed).

As illustrated in this one-node example, the multistream EKF update is not an average of the individual updates, but rather it is coordinated through the global scaling matrix $\boldsymbol{A}^*$. It is intuitively clear that this coordination is most valuable when the various streams place contrasting demands on the network.

D. Advantages and Extensions of Multistream Training

Discussions of the training of networks with external recurrence often distinguish between series-parallel and parallel configurations [13]. In the former, target values are substituted for the corresponding network outputs during the training process. This scheme, which is also known as teacher forcing, helps the network to get "on track" and stay there during training. Unfortunately, it may also compromise the performance of the network when, in use, it must depend on its own output. Hence it is not uncommon to begin with the series-parallel configuration, then switch to the parallel configuration as the network learns the task. Multistream training seems to lessen the need for the series-parallel scheme; the response of the training process to the demands of multiple streams tends to keep the network from getting too far off track. In this respect, multistream training seems particularly well suited for training RMLP's, where the opportunity to use teacher forcing is limited, because "correct" values for most if not all outputs of recurrent nodes are unknown.

Though our presentation has concentrated on multistreaming simply as an enhanced training technique, one can also exploit the fact that the streams used to provide input–output data need not arise homogeneously, i.e., from the same training task. Indeed, it is interesting to contemplate teaching a network to do multiple tasks. In contrast to feedforward networks, which can only carry out static input–output mapping, recurrent networks embed the concept of state. Hence, it is reasonable to envision that such a network might be trained to exhibit different behavior according to its current region of state space. To one extent or another, all the examples that follow make use of this capability.

A concrete expression of this idea is the use of multistreaming to coerce a network into desired behavior in conjunction with its primary task. Our first such use was explicit training of controller networks to be robust over a range of systems to be controlled. Section V-B uses the technique to induce stability on a model when its primary training objective is modeling performance when driven by a rapidly changing input. In principle, any performance objective or set of objectives that can be cast in a form that produces errors to place in $\boldsymbol{\xi}$ and derivatives that can be placed in the $\boldsymbol{H}$ matrix can be treated with the multistream method.

V. Synthetic Examples

A. Multiple Series Prediction and Dropout Rejection

In [9] and [10] we demonstrated that a single fixed-weight network could be trained to make single-step predictions for 13 different time series, including versions of the chaotic logistic, Henon, and Ikeda maps, as well as some sinusoids. In testing, the sequence presented as network input could be switched at any time from one series to another. Thus, we required the fixed-weight network to perform steps that are often associated with explicitly adaptive systems: 1) determine the identity of the time series or the parameters of the generating equation from the recent time history of the series; 2) instantiate this information into the prediction system and begin making predictions; and 3) monitor the success of predictions and, as necessary, reevaluate the series identity or its parameters.

In this section, we present an extension of this idea in which we not only consider multiple series but also allow the input sequence to be severely disrupted in the form of dropouts, i.e., input values of zero. We use the following two variations of the Henon map. (Note that Variation B is a sign-reversed version of Variation A.)

$\mathtt{Variation\ A:}$

$$x_a(t+1) = 1 - 1.4x_a^2(t) + 0.3x_a(t-1) \tag{32}$$

$\mathtt{Variation\ B:}$

$$x_b(t+1) = -1 + 1.4x_b^2(t) + 0.3x_b(t-1). \tag{33}$$

1) Training: We prepared five data sets for training, 1000 points each, for each of the two series; the sequences for two variations were prepared with dropout probabilities 0, 0.1, 0.2, 0.3, and 0.4. No limit on the number of consecutive

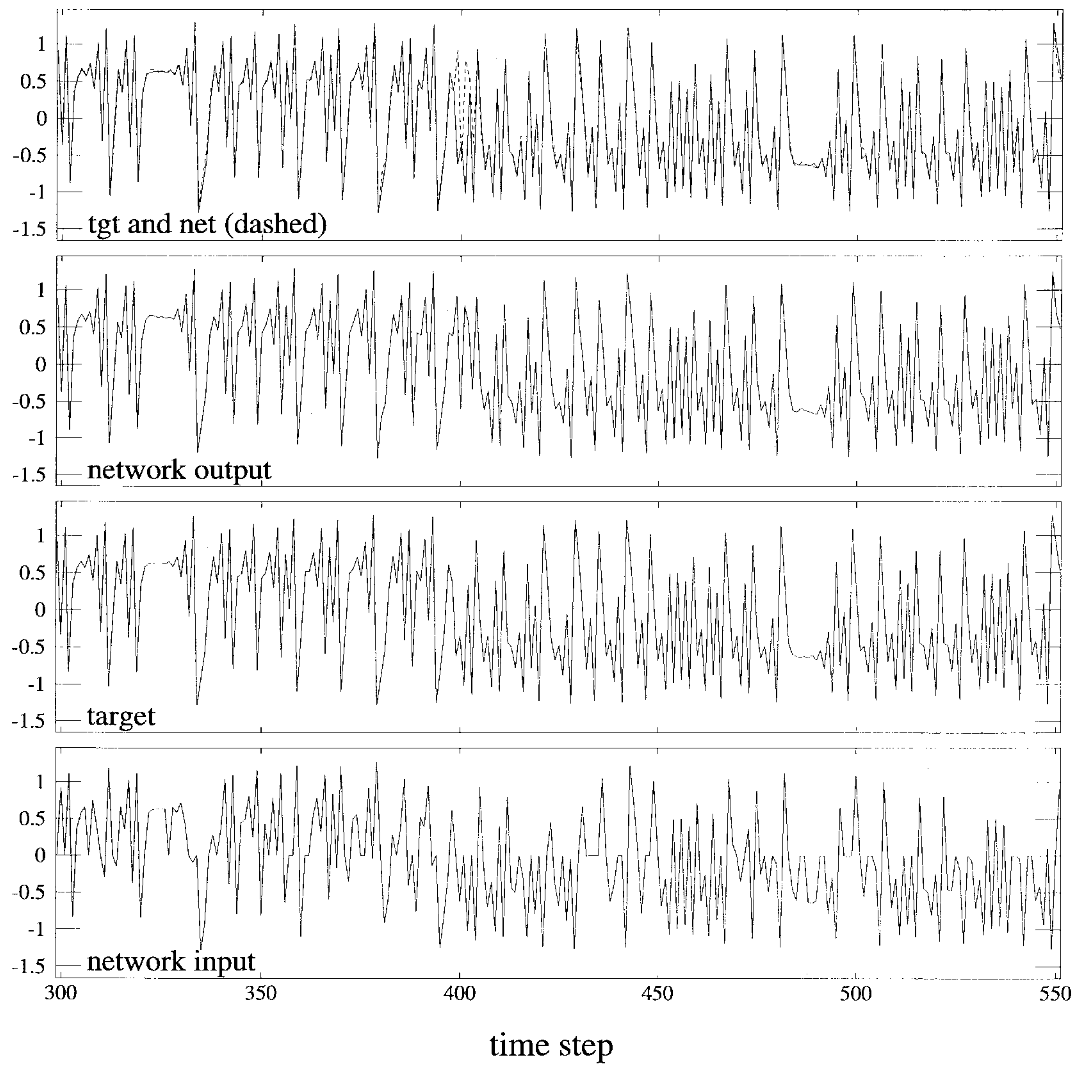


Fig. 3. A portion of a testing sequence to illustrate the performance of a network trained to handle dropouts in the input sequence and to make one-step predictions for two variations of the Henon map. For this sequence, the dropout probability is 0.3. The panel labeled "network input" shows the effect of dropouts (input values of 0); in the absence of dropouts, this panel would be identical with the panel labeled "target," apart from a one-step offset in time. An abrupt switch from variation A to variation B occurs at step 400. Network prediction errors are quite small except just after the switch.

dropouts was imposed. All sequences are initialized with randomly chosen values for $x(t)$ and $x(t-1)$; transient behavior was minimized by running the generating program for 100 time steps before recording the input–output pairs used for training.

We trained an RMLP with structure 1-5R-5R-1L, i.e., one input, two hidden layers with five fully recurrent nodes each (bipolar sigmoid activation), and a linear output node. We employed multisteam GEKF using ten streams, a priming length of ten, and a trajectory length of 200. A truncation depth $h = 10$ for BPTT was used. The network was trained for 550 $\times$190 updates, based on 550$\times$10$\times$200 input–output pairs. At termination of training the RMS error was approximately 0.07, which is about one-tenth of the standard deviation of each series. The utility of multistreaming is illustrated by noting that this error is about half that obtained with simple presentation-order scrambling, as described in Section IV-B. This was carried out using the same procedure as above except that a single-stream is used and the priming and trajectory lengths are $N_p = 19$ and $N_s = 20$, respectively, and 1 100 000 weight updates were performed (so that training would based on a comparable number of instances).

Considered as a signal processing problem, the challenges include: 1) accommodation to the current time series; 2) recognizing the difference between corrupt and good input values; and 3) combining input signals with information stored as network states to form a prediction. Because of the existence of two series, the network must pay attention to the input sequence in order to determine which series is active. Further, because both series are chaotic, even the simpler problem of making predictions for a single series without dropouts requires use of input information to keep the prediction on track.

2) Testing: To test the performance of the trained network we generated new data sequences in which we switch between series. We generated sequences with dropout probabilities 0.3 and 0.5. In Fig. 3, we show a portion of this

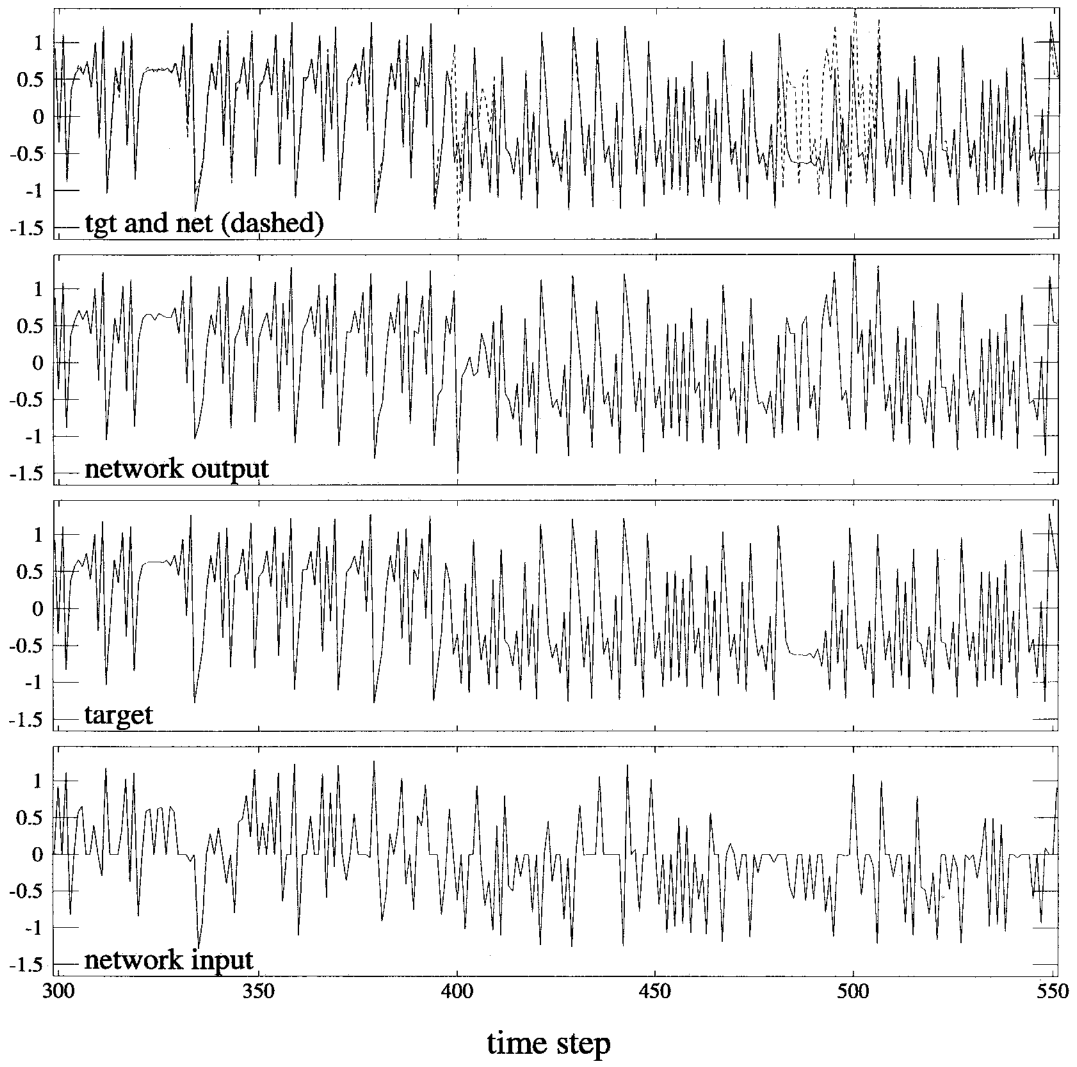


Fig. 4. Same as Fig. 3, except that the dropout probability has been increased to 0.5. Initialization of the underlying series was identical to that in Fig. 3. Again, network prediction errors increase just after the switch from one series variation to the other. In addition, the chance occurrence of a large number of dropouts following step 475 causes a momentary loss in the network's prediction ability.

sequence with dropout probability 0.3. In spite of corrupt input data, the network output is very close to the desired outputs for both series. At step 400, the desired output is calculated from (33) for series B, using the two previous outputs of series A; having thus been initialized from a state which is not on its attractor, the chaotic time series requires a number of steps to "stabilize." The network suffers a momentary loss of prediction accuracy as it accommodates the new series, but by step 409 it is again making good predictions, even in the face of continuing random dropouts. As we demonstrated in [9], multistream training allows one to sacrifice overall accuracy in favor of rapid accommodation by reducing the priming length.

When the dropout probability is set to 0.5, as in Fig. 4, the input sequence is substantially more corrupt. (Note the difference between the lower two panels.) The switch between series is handled fairly gracefully, but the chance occurrence of many dropouts between steps 470 and 500 gives rise to substantial prediction errors, which subside as the frequency of dropouts decreases. These prediction errors should not be surprising since this testing sequence includes dropouts that are likely to occur more frequently than experienced during training. Of course, for long enough dropout sequences the prediction problem becomes essentially impossible.

3) Discussion: It is interesting to contemplate how one might write a computer program to solve this problem, given complete information. In general terms, we might proceed as follows: 1) program each of the generating series so as to compute the next sequence value from the proceeding two values; 2) compare the actual observed value with the two generated values to determine which series is active; 3) issue a prediction for the next value based on the determined series.

One might also attempt to make predictions using a feedforward network with inputs based on the present input and a number of previous input values. This approach is based on the assertion that a sufficient number of input values defines a lag space, a point of which can be mapped uniquely to the desired output value. The existence of dropouts complicates matters, since missing values require that a prediction be based on a subspace of the full lag space. Further, the required subspace depends on the positions in which missing values occur in the input sequence.

In order to test whether this problem is easily solved using a modest number of lagged inputs, we carried out training using a feedforward network whose inputs came from a tapped delay line of eight elements (the current input and seven previous values). The network structure was 8-10-10-1L. The same procedure as used to train the recurrent network was employed. Training for an equivalent number of updates produced an RMS error of 0.34, which is substantially inferior to the value of 0.07 reported above. Though this experiment certainly does not prove that mapping from a lag space cannot handle this problem, it confirms our suspicion that even if such a mapping exists, it might be very difficult to learn.

B. Modeling with Stability Enforcement

In [1], Suykens *et al.* consider the problem of training a time-lagged recurrent neural network to identify the model parameters of an internally stable system that is corrupted by process noise. In that work, they demonstrated that training of a neural network model for the considered system via application of dynamic backpropagation can result in a model that exhibits undesirable limit cycle behavior when operated autonomously. Then they established that a neural network model can be trained such that global asymptotic stability is enforced by modifying dynamic backpropagation as in [13] to include stability constraints. Here, we show that the multistream training procedure provides an alternative mechanism that allows a neural network model to be developed which satisfies the modeling requirements of the forced system while simultaneously coercing stable autonomous behavior via the use of an auxiliary training data stream.

The single-input/single-output system proposed in [1] is given by

$$x_{k+1} = W_{AB}\tanh(V_A x_k + V_B u_k) + \phi_k \tag{34}$$

$$y_k = W_{CD}\ \tanh(V_C x_k + V_D u_k). \tag{35}$$

In these equations, ϕ_k refers to zero-mean, normally distributed process noise, x_k is a 4-component vector that encodes the state of the system, y_k is the observable system output, u_k is the system input, and $W_{AB}, W_{CD}, V_A, V_B, V_C$ and V_D encode the model parameters, given by

$$W_{AB} = \begin{bmatrix} 0.4157 & -0.2006 & 0.1260 & -0.0237 \\ 1.1271 & -0.0401 & -0.6084 & 0.4073 \\ -0.2141 & 0.4840 & -0.2966 & -0.0027 \\ -0.1986 & -0.6325 & 0.4208 & -1.0233 \end{bmatrix}$$

$$W_{CD} = [-0.5546 \quad -0.2603 \quad 1.3030 \quad -1.3587]$$

$$V_A = \begin{bmatrix} -0.3152 & -0.8392 & -0.2323 & -0.5119 \\ -0.2872 & -0.7385 & -0.4354 & 0.4126 \\ -0.1009 & -0.6593 & -0.5717 & -0.8109 \\ -0.8850 & -0.7005 & -0.0671 & 0.0592 \end{bmatrix}$$

$$V_C = \begin{bmatrix} 0.3600 & 0.5972 & 1.7870 & -1.4743 \\ 1.3145 & -0.2945 & 0.0347 & 0.2681 \\ -1.2125 & -0.9360 & 0.3255 & 1.7658 \\ 0.5938 & -0.2655 & 0.5651 & -1.7682 \end{bmatrix}$$

$$V_B = \begin{bmatrix} 0.1256 \\ 1.2334 \\ 1.0599 \\ -1.7554 \end{bmatrix} \quad V_D = \begin{bmatrix} 1.6275 \\ -2.3663 \\ -0.8700 \\ -0.7000 \end{bmatrix}.$$

Note that this system can be expressed as a time-lagged recurrent neural network with an ordered network representation, as shown in Table 1. This system can also be expressed as an RMLP, but this representation would require additional nontrainable nodes and weights that encode time-delay connections, as well as connections that pass the control input between layers without modification.

1) Training Without Constraints: We followed closely the problem statement as described in [1]. A training data set is derived by choosing control signals u_k at each time step from a zero-mean, normal distribution with a standard deviation of five. Similarly, the standard deviation of the process noise is set to 0.1. A set of 1000 data points was generated in this fashion, with the first 500 used for training and the last 500 for independent testing. Values of the initial state variables of the system were set to zero.

We carried out training experiments with two different network architectures. In the first case, we used a network representation of (34)–(35); this network contains 13 trainable nodes (of which four are effectively state nodes) with associated trainable weights. The second network considered is a completely trainable RMLP with structure denoted by 1-8R-7R-1L; in this case, we do not assume exact knowledge of the underlying model structure. This network has a substantially greater number of degrees of freedom and states than does the system being modeled.

We first conducted training trials for the two networks using only training data from the forced system with no constraints to enforce stability. In each case, we employed multistream global EKF training with three streams, a priming length of five steps, a trajectory length of 100 steps, and a backpropagation truncation depth of $h = 19$. Excellent results were observed for the testing portion of the generated data under conditions of rapidly changing input signal. In [1], the autonomous behavior of the neural model was compared to that of the system, but without the process noise that was present while input–output data were accumulated for network training. (In a real application, it probably would not be possible to examine the actual system behavior in the absence of process noise. For the present system model, the process noise specified is sufficient to drive the unforced $(u_k = 0)$ system to substantial output values, over unity in magnitude, making it difficult to observe whether the model exhibits stable behavior.) When the state nodes of the trained network models were initialized to random values in the range $[-1, +1]$ and then executed autonomously (i.e., with zero input), limit-cycle output behavior was observed for both networks for many different state initializations. Typical

Table 1 Ordered Network Representation of System Equations (34) and (35)[1]

i	j	$c_{i,j}$	$d_{i,j}$	W
2	1	1	0	$V_{B:1}$
	2	6	1	$V_{A:1,1}$
	3	7	1	$V_{A:1,2}$
	4	8	1	$V_{A:1,3}$
	5	9	1	$V_{A:1,4}$
3	1	1	0	$V_{B:2}$
	2	6	1	$V_{A:2,1}$
	3	7	1	$V_{A:2,2}$
	4	8	1	$V_{A:2,3}$
	5	9	1	$V_{A:2,4}$
4	1	1	0	$V_{B:3}$
	2	6	1	$V_{A:3,1}$
	3	7	1	$V_{A:3,2}$
	4	8	1	$V_{A:3,3}$
	5	9	1	$V_{A:3,4}$
5	1	1	0	$V_{B:4}$
	2	6	1	$V_{A:4,1}$
	3	7	1	$V_{A:4,2}$
	4	8	1	$V_{A:4,3}$
	5	9	1	$V_{A:4,4}$
6	1	2	0	$W_{A,B:1,1}$
	2	3	0	$W_{A,B:1,2}$
	3	4	0	$W_{A,B:1,3}$
	4	5	0	$W_{A,B:1,4}$
7	1	2	0	$W_{A,B:2,1}$
	2	3	0	$W_{A,B:2,2}$
	3	4	0	$W_{A,B:2,3}$
	4	5	0	$W_{A,B:2,4}$
8	1	2	0	$W_{A,B:3,1}$
	2	3	0	$W_{A,B:3,2}$
	3	4	0	$W_{A,B:3,3}$
	4	5	0	$W_{A,B:3,4}$
9	1	2	0	$W_{A,B:4,1}$
	2	3	0	$W_{A,B:4,2}$
	3	4	0	$W_{A,B:4,3}$
	4	5	0	$W_{A,B:4,4}$
10	1	1	0	$V_{D:1}$
	2	6	1	$V_{C:1,1}$
	3	7	1	$V_{C:1,2}$
	4	8	1	$V_{C:1,3}$
	5	9	1	$V_{C:1,4}$
11	1	1	0	$V_{D:2}$
	2	6	1	$V_{C:2,1}$
	3	7	1	$V_{C:2,2}$
	4	8	1	$V_{C:2,3}$
	5	9	1	$V_{C:2,4}$
12	1	1	0	$V_{D:3}$
	2	6	1	$V_{C:3,1}$
	3	7	1	$V_{C:3,2}$
	4	8	1	$V_{C:3,3}$
	5	9	1	$V_{C:3,4}$
13	1	1	0	$V_{D:4}$
	2	6	1	$V_{C:4,1}$
	3	7	1	$V_{C:4,2}$
	4	8	1	$V_{C:4,3}$
	5	9	1	$V_{C:4,4}$
14	1	10	0	$W_{C,D:1,1}$
	2	11	0	$W_{C,D:1,2}$
	3	12	0	$W_{C,D:1,3}$
	4	13	0	$W_{C,D:1,4}$

limit-cycle behavior of the trained network with structure identical to that of the system model is shown in Fig. 5 for three different state initializations. It is noteworthy that limit cycles of different peak-to-peak magnitudes as well as of different periods are observed as the result of different training trials (e.g., different initial conditions and orders of presentation of training data).

2) Training with Constraints: In order to enforce stability of the autonomous system by training we must be able to

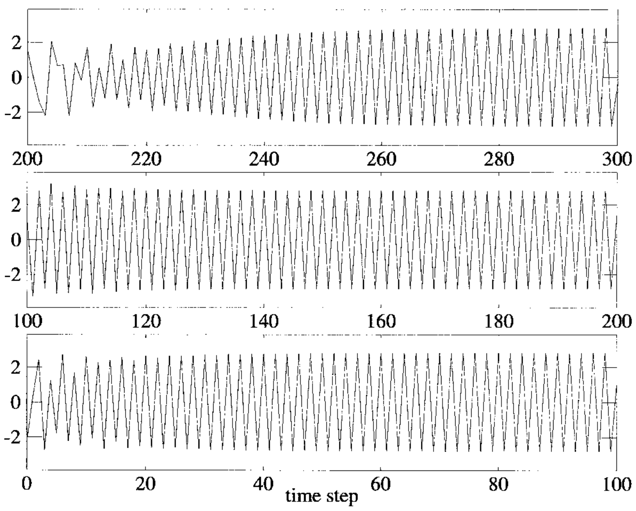


Fig. 5. Autonomous behavior of time-lagged network with structure identical to that of the system model trained without stability enforcing constraints. Random state initializations occur at time steps 0, 100 and 200. The unforced network model exhibits limit cycle behavior.

provide the training process with network inputs and target outputs. The most obvious procedure would be to use the observed autonomous behavior of the stable system we are modeling. In the present case, however, we cannot rely on the system itself to provide a reasonable training set because we cannot use the behavior of the system in the absence of process noise without violating the premise of the problem. Thus we took a different approach in which we synthesized a secondary training set of 500 points, where each training pattern consists of zero input and zero output. This reflects the goal that the network model should exhibit asymptotic stability. Then we employed multistream training to model both the forced system, using the data discussed earlier, and the desired stable behavior of the unforced system, as seen through the synthesized data sequence. We employed six training streams altogether, evenly divided between forced and autonomous behaviors. Trajectory lengths of 100 were used, with priming lengths of five time steps. Derivatives were computed by BPTT(19) and global multistream EKF weight updates were applied. This example is unique among those considered in this paper in that, during training, we instantiate the network with specific values of the state variables, chosen randomly, at the beginning of each training trajectory for both forms of data. A total of 120 000 instances were processed during training of each network.

We show representative autonomous behavior for multiple random state initializations for the two different network architectures in Figs. 6 and 7. Note that the outputs of both network models appear to approach zero asymptotically. This behavior is expected for the case of the network model that is structured identically to the original system, since this model has no bias connections. On the other hand, the 1-8R-7R-1L network has a bias connection for each trainable node; it was somewhat surprising to observe stable fixed points of magnitude less than 10^{-4} for this network, given that the bias connections must be trained so that they

[1]The elements of the input and output arrays are $I_1 = 1$ and $O_1 = 14$. Nodes 2–5 and 10–13 have nonlinear activation functions (tanh(·)), while nodes 6–9 and 14 are linear. It should be noted that this representation is not unique; alternative equivalent ordered network representations are possible.

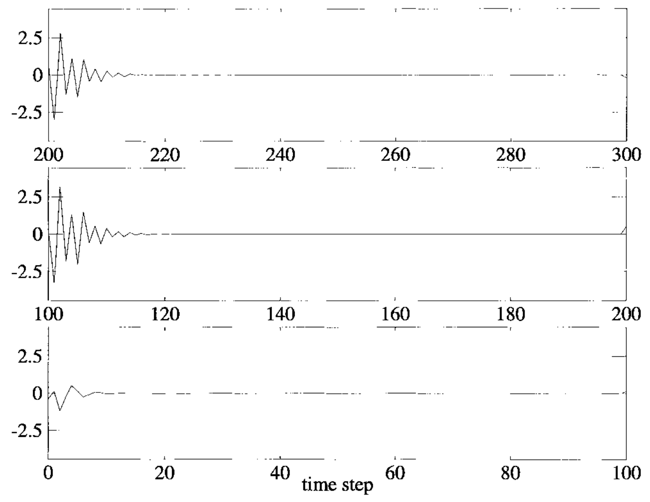


Fig. 6. Autonomous behavior of time-lagged network with structure identical to that of the system model trained with multistream stability enforcing constraints. Random state initializations occur at time steps 0, 100, and 200. Note that the model quickly reaches stable behavior, unlike the network trained without stability enforcing constraints, as shown in Fig. 5.

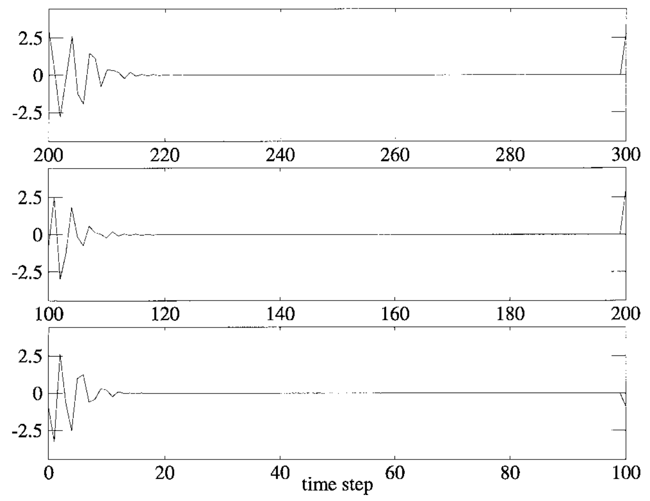


Fig. 7. Autonomous behavior of 1-8R-7R-1L RMLP trained with multistream stability enforcing constraints. Random state initializations occur at time steps 0, 100, and 200. Even though this network model is of greater complexity than the system model, stable behavior is still easily achieved.

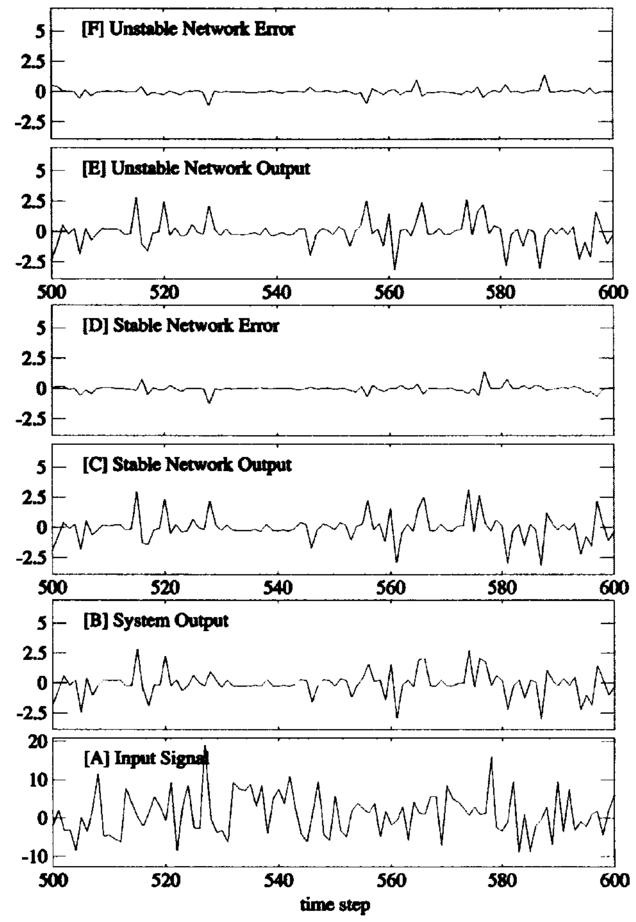


Fig. 8. Driven response of the 1-8R-7R-1L RMLP trained with multistream stability enforcing constraints compared to that of the same network trained without regard to stability. A sequence of 100 time steps from the noise-corrupted testing set is used. [A] shows the input signal, while [B] shows the noise-corrupted system output. Network responses to the input pattern are shown in [C] (stability enforced) and [E] (stability ignored), respectively. Network errors for these two cases are shown in [D] and [F], respectively. Note that the networks perform comparably.

effectively cancel one another under autonomous operation. Finally, Fig. 8 demonstrates that the forced behavior of the 1-8R-7R-1L network that is trained for stable behavior is not significantly compromised relative to the forced behavior of the network that was not trained to be stable; qualitatively similar and comparably accurate behavior is observed for the network that has structure identical to that of the system.

3) Discussion: It is important to note that use of the multistream method to enforce stability affords us some flexibility in tailoring the behavior of the autonomous network as it converges toward an output value of zero. By setting the priming length N_p, we effectively specify a number of steps from the beginning of each training trajectory during which we do not demand a particular network output. In the case of stability enforcement, a larger value of the priming length is expected to lead to less rapid convergence to zero, since the cost function penalizes nonzero output of the autonomous network only for time steps beyond the priming length. In the present case, the convergence of the autonomous model is less rapid than that of networks trained with $N_p = 5$, suggesting that a network model should be trained with a larger priming length. We found that taking $N_p = 25$ and retraining led to autonomous behavior that was closer to that of the actual system. In practice, of course, one would probably not be able to turn off the process noise so as to obtain a convergence standard. Thus it may be possible only to specify a desired convergence characteristic for the network model and select the priming length accordingly.

VI. Automotive Examples

The general area of automotive powertrain control and diagnosis offers substantial opportunity for application of intelligent signal processing methodologies. These opportu-

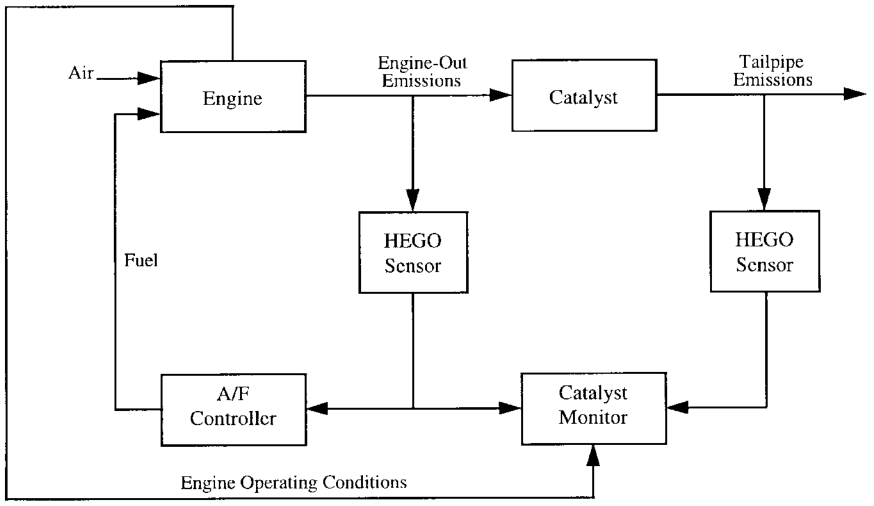


Fig. 9. A block diagram of components of a typical catalyst monitoring scheme in the presence of a feedback A/F control system.

nities are driven by the steadily increasing demands that are placed on the performance of vehicle control and diagnostic systems as a consequence of global competition and government mandates. Modern automotive powertrain control systems involve several interacting subsystems, any one of which can involve significant engineering challenges. Further, increasingly stringent emissions regulations require that any malfunctioning component or system with the potential to undermine the emissions control system be detected and identified. In this section, we consider two signal processing problems related to automotive diagnostics that are particularly amenable to treatment with recurrent neural networks trained by the multistream EKF formalism.

A. Sensor-Catalyst Modeling

A particularly critical component of a vehicle's emissions control system is the catalytic converter. The role of the catalyst is to chemically transform noxious and environmentally damaging engine-out emissions, which are the by-product of the engine's combustion process, to environmentally benign chemical compounds. In particular, an ideal three-way automotive catalytic converter should completely perform the following three tasks during continuous vehicle operation: 1) oxidation of hydrocarbon (HC) exhaust gases to carbon dioxide (CO_2) and water (H_2O); 2) oxidation of carbon monoxide (CO) to carbon dioxide; and 3) reduction of nitrous oxides (NO_x) to nitrogen (N_2) and oxygen (O_2). In practice, it is possible to achieve high catalytic conversion efficiencies simultaneously for all three types of engine-out exhaust gases only for a very narrow window of operation of air/fuel ratio (A/F); when this occurs, the engine is operating near stoichiometry.

A major role of the engine control system is to regulate A/F about stoichiometry. This is accomplished with an electronic feedback control system that utilizes a heated exhaust gas oxygen (HEGO) sensor whose role is to indicate whether the engine-out exhaust is rich (i.e., too much fuel) or lean (too much air). Depending on the measured state of the exhaust gases, the A/F control is changed so as to drive the system toward stoichiometry. Since the HEGO sensor is largely considered to be a binary sensor (i.e., it produces high/low voltage levels for lean/rich operations, respectively), and since there are time-varying transport delays, the closed-loop A/F control strategy often takes the form of a jump/ramp strategy, which effectively causes the HEGO output to oscillate between the two voltage levels.

Even in the presence of an effective closed-loop A/F control strategy, vehicle-out (i.e., tailpipe) emissions may be unreasonably high if the catalytic converter has been damaged. For example, the catalytic converter may degrade due to exposure to excessively hot temperatures (as may be the case due to frequent misfires, as we describe below). We would also expect the performance of the catalytic converter to degrade with continued use. Current government regulations require that the performance of a vehicle's catalytic converter be continuously monitored to detect when conversion efficiencies have dropped below some threshold. Unfortunately, it is currently infeasible to equip vehicles with sensors that can measure the various exhaust gas species directly. Instead, current catalytic converter monitors are based on comparing the output of a HEGO sensor that is exposed to engine-out emissions to the output of a second sensor that is mounted downstream of the catalytic converter and is exposed to the tailpipe emissions, as illustrated schematically in Fig. 9. This approach is based on the observation that the postcatalyst HEGO sensor switches infrequently, relative to the precatalyst HEGO sensor, in the presence of a highly efficient catalyst. Similarly, the average rate of switching of the postcatalyst sensor increases as catalyst efficiency decreases (due to decreasing oxygen storage capability).

As an alternative, a model-based catalyst monitor can be envisioned where the actual output of the postcatalyst HEGO sensor is compared to that of a model of the postcatalyst sensor that assumes a catalytic converter with some nominal converter efficiency. Then, based upon differences

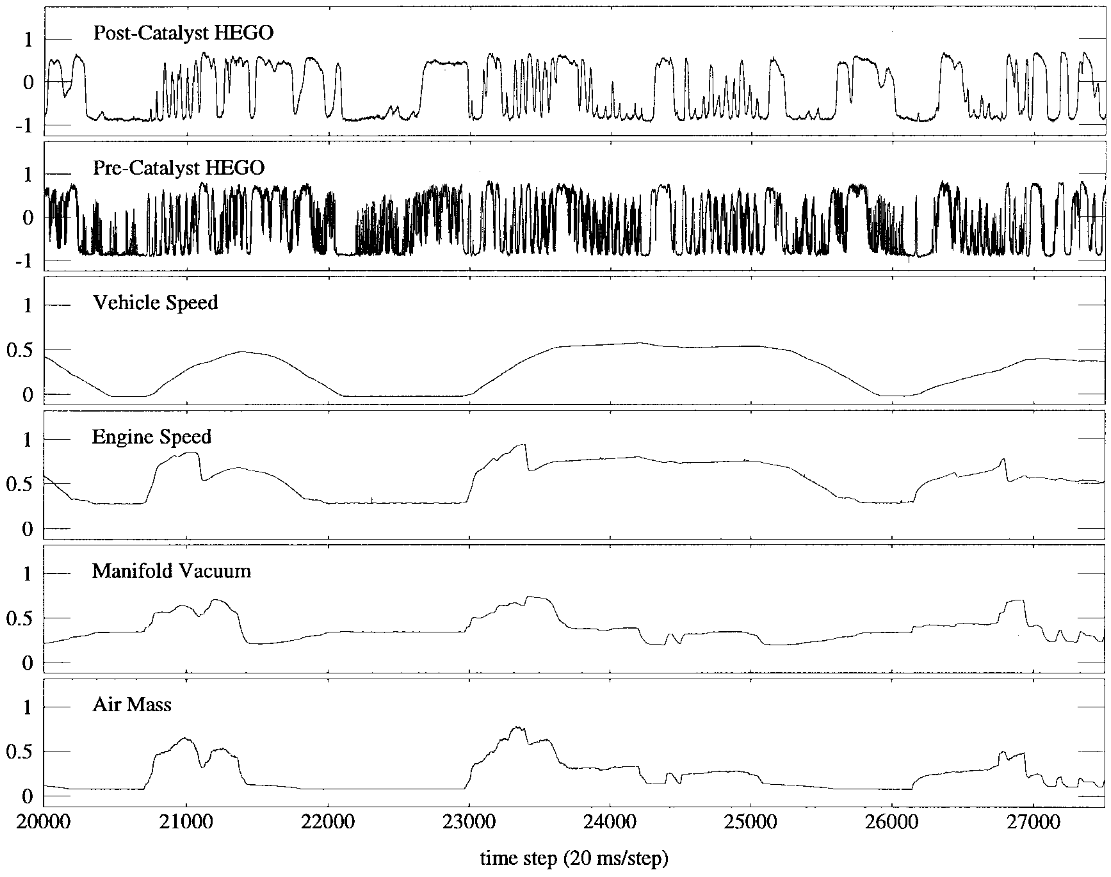


Fig. 10. A 150 s segment of training data for the catalyst-sensor model. All variables are shown in scaled units.

between the actual sensor and model outputs, a procedure can be developed to estimate catalytic converter efficiency. However, this modeling is by no means a trivial task. First, the dynamical behavior of catalytic converters, as well as that of HEGO sensors, is not completely understood from a physical and chemical perspective. Second, conditions that affect catalyst and sensor performance are often not observed. Third, the model output must incorporate condition-dependent time delays, e.g., the transport delay associated with the physical placement of the sensors as well as catalyst activity is seen to be a function of engine speed. Here we describe how a time-lagged recurrent neural network can be trained with multistream training methods to represent such a system, using only information that could be made available to the vehicle's powertrain control module.

1) Experimental Data: Data were acquired from a single vehicle equipped with a thermally degraded catalyst. A standard driving cycle was employed to obtain training and testing data of the operating vehicle on a chassis rolls dynamometer facility. Relevant engine variables were sampled at 20 ms intervals. Amongst the variables sampled were vehicle speed, engine speed, air mass, manifold vacuum, precatalyst HEGO sensor output, and postcatalyst sensor output. Two data acquisition runs were performed, on different days. Each run had 78 000 data points and required a time of 26 min. A representative sample of data is shown in Fig. 10. Here we see that there is very slow variation of the air mass, manifold vacuum, engine speed and vehicle speed signals, whereas the precatalyst sensor exhibits rapid switching behavior that is somewhat more frequent than that of the postcatalyst sensor. It is noteworthy that although the HEGO sensor output is often considered to be binary in nature, in fact it has an analog nature in which switches do not appear to occur instantaneously and in which the voltage levels reached are not distinctly binary in nature. Analysis of the data discloses that the time delay between the precatalyst and postcatalyst sensor outputs varies from less than 0.1 s at high engine speed/air mass combinations to more than 1 s at low engine speed/air mass combinations. Finally, depending on vehicle operating conditions, the postcatalyst HEGO sensor output sometimes closely mirrors the switching characteristics of the precatalyst sensor, while at other times it appears to be largely independent of the precatalyst sensor output.

2) Training and Testing: A time-lagged recurrent neural network was trained on one of the data sets for which the average HC conversion efficiency was measured to be nearly 80.0%. Only 63 000 of the 78 000 data points gathered were actually used for training; the beginning section corresponding to cold start (5000 points) was ignored, as were data acquired after the engine was turned off. The network inputs at any time step are given by the current air mass, manifold vacuum, engine speed, and vehicle speed. In addition, a sparse tapped delay line representation of the

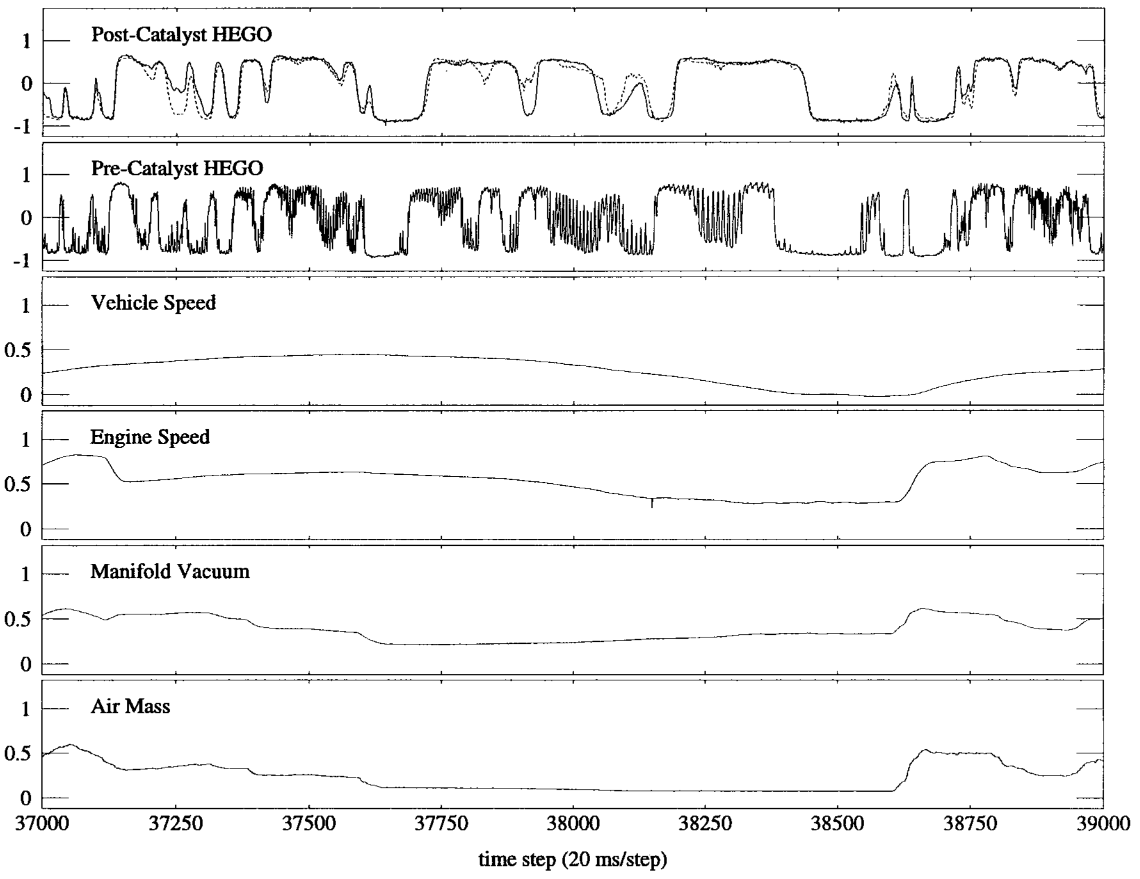


Fig. 11. A 40 s segment of testing data for catalyst-sensor model. The top panel shows the actual postcatalyst HEGO sensor output in a solid pattern and the predicted HEGO sensor output in a dashed pattern.

precatalyst HEGO sensor output was formed as input to the network, consisting of the current measurement along with ten additional measurements of the sensor output spaced five time steps (0.1 s) apart, spanning a total time of 1 s. The resulting input vector at any time step is comprised of 15 signals plus a bias. The network architecture chosen was an RMLP with structure given by 15-20R-15R-10R-1; the resulting network consisted of 1531 weights. Simpler architectures were found not to be as effective. (We have found the combination of sparse tapped delay line input representations with internal network recurrence to be a particularly effective mechanism for treating the problem, highlighted in [25], of learning long-term dependencies and condition dependent time delays.) Multistream training was performed utilizing ten data streams, with trajectory lengths of 1000 instances and priming lengths of 50 time steps. Derivatives were computed by backpropagation through time with a truncation depth of 74 time steps. Due to the complexity and size of the network architecture, node-decoupled multistream weight updates were performed. This procedure resulted in the processing of 2.2 million instances during training, with each instance processed an average of 35 times.

We performed testing of the trained network with the second data set obtained from the same vehicle/catalyst combination. Interestingly, the average HC conversion efficiency for this data set was measured to be 75%, a 5% difference from the training set (it is possible that most of the difference could be attributed to the cold-start portion, which we have ignored for the present purposes). Fig. 11 shows a representative sequence of engine operating conditions and network behavior over 40 s; this sequence includes both low- and high-speed operations, as well as some transients due to vehicle accelerations and decelerations. It is evident that the trained network has captured the qualitative behavior of the postcatalyst HEGO sensor output. In Figs. 12–14, we show 5 s portions of the network performance for different operating conditions. Fig. 12 demonstrates network performance for the vehicle operating largely at idle, where the time delay between precatalyst and postcatalyst sensor outputs is expected to be a maximum. Alternatively, Fig. 13 shows results for vehicle acceleration and high engine speed operation, where we expect to see a minimal time delay. Finally, Fig. 14 demonstrates 5 s of network performance under conditions of medium speed cruise. With the exception of some small phase shifts and amplitude deviations, the network output appears to closely follow that of the actual postcatalyst HEGO sensor.

B. Engine Misfire Detection

Engine misfire is broadly defined as the condition in which a substantial fraction of a cylinder's air-fuel mixture fails to ignite. Frequent misfires will lead to a deterioration of the catalytic converter, ultimately resulting in

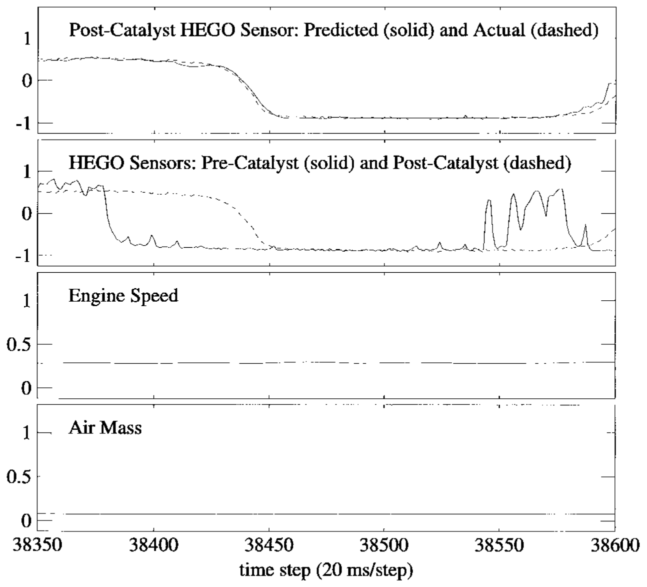


Fig. 12. A 5 s segment of network performance for the vehicle operating at idle. For compactness, we have eliminated the manifold vacuum and vehicle speed traces (see Fig. 11) for the corresponding segments. In the second panel from the top, the precatalyst and postcatalyst HEGO sensor outputs are plotted together to provide a visual sense of the context dependent time delay. In the uppermost panel, the postcatalyst HEGO sensor output is repeated as a dashed pattern, and the network output is shown as a solid line. Note that the network output is able to closely capture the dynamic relationship associated with the long time delays between the pre and postcatalyst sensor outputs.

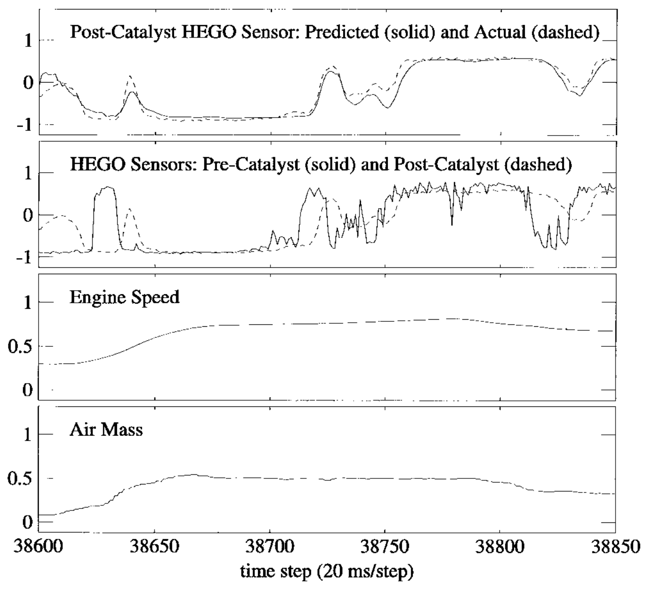


Fig. 13. A 5 s segment of network performance for the vehicle accelerating to a high speed, followed by a deceleration. The panels are arranged identically to Fig. 12. Note that the network gracefully handles the context transients, and that the short time delays between the two sensors are properly modeled.

unacceptable levels of emitted pollutants and a costly replacement. Consequently, government mandates require that automotive manufacturers provide on-board misfire detection capability under nearly all engine operating conditions for vehicles sold in the United States after 1998.

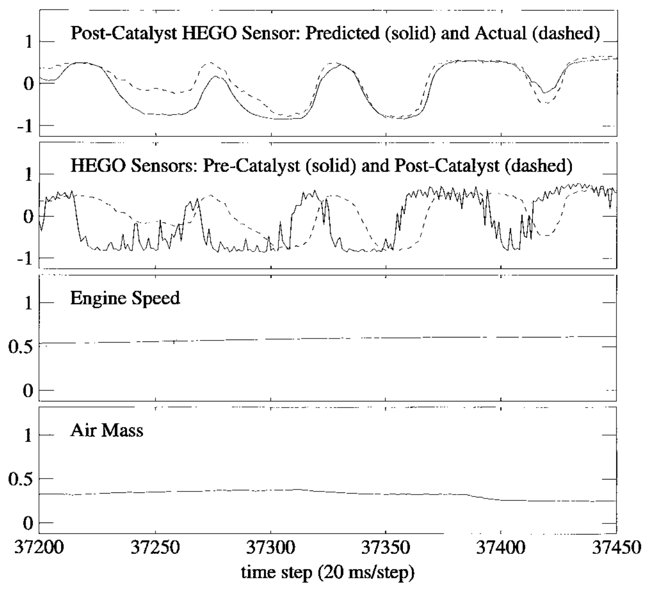


Fig. 14. A 5 s segment of network performance for the vehicle operating at a relatively constant speed. The panels are arranged identically to Fig. 12. Note that the time delays in this case are of moderate length.

While there are many ways of detecting engine misfire, all currently practical methods rely on observing engine crankshaft rotational dynamics with a position sensor located at one end of the shaft. Briefly stated, one looks for a crankshaft acceleration deficit following a cylinder firing and attempts to determine whether such a deficit is attributable to a lack of power provided on the most recent firing stroke. (In effect, the momentary unopposed load on the engine causes it to slow down briefly.)

Since every engine firing must be evaluated, the natural "clock" for misfire detection is based on crankshaft rotation, rather than on time. For an n-cylinder engine, there are n engine firings, or events, per engine cycle, which requires two engine revolutions. The actual time interval between events varies considerably; for an eight-cylinder engine, for example, the time interval varies from 20 ms at 750 revolutions per minute (RPM) to 2.5 ms at 6000 RPM. Engine speed, as required for control, is typically derived from measured intervals between marks on a timing wheel. As used in misfire detection, an acceleration value is calculated from the difference between successive intervals. If the timing marks are favorably placed, each such computed acceleration depends sensitively on the firing of just one cylinder.

Fig. 15 shows a sequence of acceleration values, taken when the engine is at low speed and lightly loaded. Misfires have been artificially induced (by interrupting the spark), and acceleration values that correspond to misfires have been labeled in the plot. Under these conditions, misfires are easily detected by simple thresholding.

This scheme is complicated by several factors. One of these is that the angular intervals between timing marks may not be precisely equal and may differ from vehicle to vehicle. This is typically handled by an adaptive correction

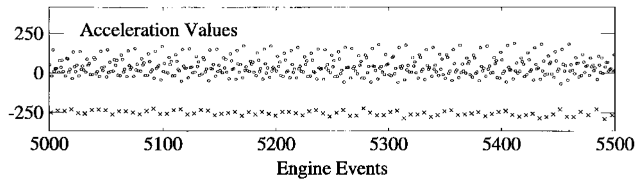


Fig. 15. A temporal sequence of acceleration values (scaled units) for low-speed engine operation. Artificially induced misfires are denoted by "×" symbols.

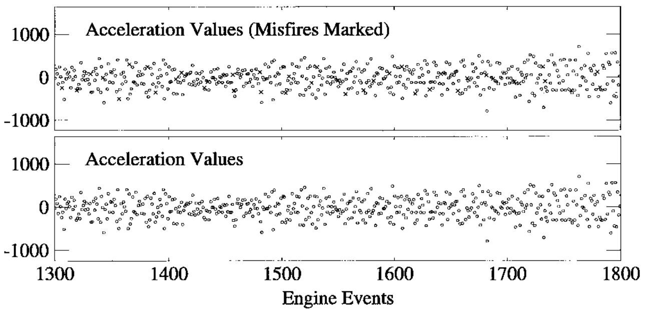


Fig. 16. A temporal sequence of acceleration values, illustrating the effects of crankshaft dynamics. In the upper panel misfires are denoted by "×" symbols.

that is carried out on board each vehicle; such a correction has been applied to all data shown here. A more serious problem is that the crankshaft is not infinitely stiff. This causes it to exhibit complex torsional dynamics, even in the absence of misfire. The magnitude of acceleration induced by such torsional vibrations may be large enough to dwarf acceleration deficits from misfire. Further, the torsional vibrations are themselves altered by misfire, so that normal engine firings following a misfire may be misinterpreted.

In the lower panel of Fig. 16 we have again plotted an acceleration sequence, but under the more challenging conditions of high speed and moderately high load. It is essentially impossible to pick out misfires visually. In the upper panel we have labeled the misfires; note how they are mixed with normal values.

Although the effect of torsional oscillation is to add what may appear to be random noise to the sequence of acceleration values, we knew from prior work of colleagues that the pattern, in the absence of misfire, was fairly reproducible at a given operating condition, although it varied as the operating condition changed. Hence, we speculated that the problem might yield to a suitable unraveling of the acceleration signal. In effect, our approach is to use a training process to form a context-dependent nonlinear filter.

1) Applying Recurrent Networks: We have approached the misfire detection problem with recurrent networks in two related ways. In the first of these, we attempt to convert the observed acceleration sequence into a good approximation of an idealized sensor; here the latter amounts to laboratory quality time measurement on a timing wheel which is placed in a crankshaft location less subject to torsional vibration. Details of the first use of this method are presented in [26]. Performing misfire classification with this approach requires the same final steps employed in current production misfire detection. Typically, one forms the difference between each sensor value and a central measure of its temporal neighbors and normalizes the result according to the acceleration deficit expected for that engine state. The resulting value is then compared to a chosen threshold to effect the classification.

In the second approach [27] the final steps may be bypassed by training a network to perform the classification directly. Here we choose the network architecture to be 4-15R-7R-1. The inputs are engine speed, engine load (a derived quantity that is based primarily on air mass), acceleration (as described above), and a binary flag to identify the beginning of the cylinder firing sequence. The target is either $+1$ or -1, according to whether a misfire had been artificially induced for the current cylinder during the previous engine cycle. This phasing, $\tau_1 = -8$ in the notation of Section II, enables the network to make use of information contained in measured accelerations that follow the engine event being classified. The consequently noncausal nature of the nonlinear filter presents no practical problem, since the classifications are used statistically rather than for immediate action.

An important practical aspect is that, unlike the method which depends on targets based on an idealized sensor, the acquisition of data for training does not require special equipment beyond that used to extract information from the engine computer.

In the example shown here, the training database was acquired on a production vehicle with an eight-cylinder engine over a wide range of operation, including engine speed and load combinations beyond those encountered in normal driving. Misfires were deliberately induced at various intervals. Although the database consists of more than 600 000 examples (one per cylinder event), it only approximates full coverage of the space of operating conditions and possible misfire patterns. It is therefore important to test the network's generalization by applying it to independent data.

In Fig. 17 we display the same segment of data as in Fig. 16, but here we have also plotted network inputs corresponding to engine speed and load and have shown the output of the trained recurrent network. Both the acceleration values and the network output values have been labeled according to whether misfire had been induced. It should be noted that this segment is not part of the training set, but rather is part of a test set acquired after the network had been trained. The network is making remarkably few classification errors. Most network errors occur during moments of rapid positive or negative acceleration, not shown here. (Actually, misfire detection is not required during conditions of negative engine torque, such as during deceleration, when the engine is not producing power.)

VII. Summary and Conclusions

In this paper we have presented a set of techniques that, used together, have allowed us to address a range of interesting and potentially useful training problems. We

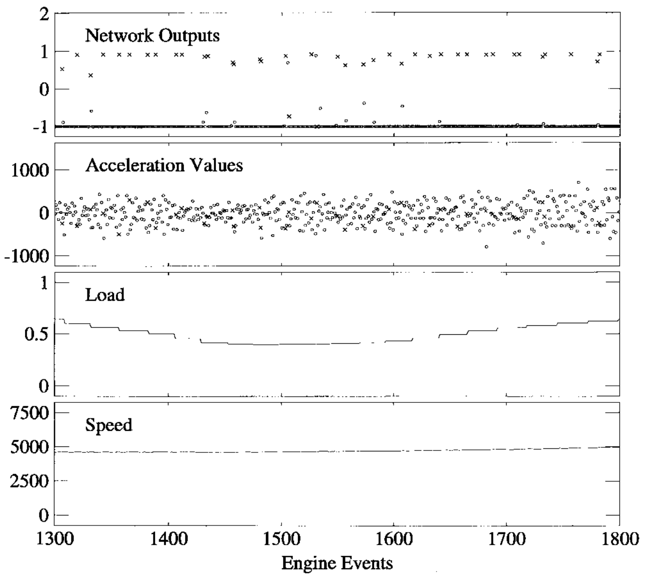


Fig. 17. The segment of acceleration values of Fig. 16 is plotted, together with inputs to a recurrent classification network and the output of the trained network. An additional network input assumes a value of one every eight steps and is zero otherwise. Misfires are again denoted by "×" symbols.

have used substantially the same procedure for each of the examples presented as well as for many others. The RMLP has served us very well, with architectural details chosen largely according to how hard a given problem is thought to be, with due regard for the amount of available data. (We have no solution to the occasionally encountered problem of a difficult mapping supported by little data.) For most problems, the networks we train are probably of a higher order than would minimally be required. More study is required to determine under what conditions this may be problematic. In any case, if the minimal architecture for a given problem is known, then our framework may be applied to it.

Though we have concentrated on off-line applications, it is entirely possible to employ EKF updates in real time, as we demonstrated for controller training in [12]. With sufficient computational power, multistream methods can also be incorporated into real-time applications. A quite promising scenario is that of combining a primary data stream based on real-time data with one or more secondary streams synthesized, as in Section V-B, to encourage desired secondary behavior such as stability or steady-state accuracy.

Some of the networks we have discussed here require significant computation merely to execute. As this could be a stumbling block to production applications, a collaboration was initiated between Ford Research and the Jet Propulsion Laboratory resulting in an elegant and inexpensive hardware implementation [28] that allows fast recurrent network execution.

ACKNOWLEDGMENT

The authors are pleased to acknowledge the various important contributions of their colleagues to this effort. In particular, they are indebted to K. Marko, J. James, and T. Feldkamp of Ford Motor Company's Advanced Diagnostics Group for inviting the authors' participation in and sharing their critical insights on the misfire detection problem. They are also grateful to J. Jesion for his many technical contributions to the experimental portions of this work, including training of the misfire classification network whose results are shown here.

REFERENCES

[1] J. A. K. Suykens, J. Vandewalle, and B. L. R. De Moor, "NL$_q$ theory: Checking and imposing stability of recurrent neural networks for nonlinear modeling," *IEEE Trans. Signal Processing*, vol. 45, pp. 2682–2691, Nov. 1997.
[2] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1994.
[3] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford, UK: Oxford, 1995.
[4] R. J. Williams, "Adaptive state representation and estimation using recurrent connectionist networks," in *Neural Networks for Control*, W. T. Miller III, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT, 1990, pp. 97–114.
[5] J. T.-H. Lo, "System identification by recurrent multilayer perceptrons," in *Proc. World Congr. Neural Networks*, Portland, OR, 1993, pp. IV-589–IV-600.
[6] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.
[7] N. E. Cotter and P. R. Conwell, "Fixed-weight networks can learn," in *Proc. Int. Joint Conf. Neural Networks*, San Diego, CA, 1990, vol. III, pp. 553–559.
[8] ——, "Learning algorithms and fixed dynamics," in *Proc. Int. Joint Conf. Neural Networks*, Seattle, WA, vol. I, 1991, pp. 799–804.
[9] L. A. Feldkamp, G. V. Puskorius, and P. C. Moore, "Adaptive behavior from fixed weight networks," *Information Sciences*, vol. 98, pp. 217–235, 1997.
[10] ——, "Adaptation from fixed weight dynamic networks," in *Proc. IEEE Int. Conf. Neural Networks*, Washington DC, 1996, pp. 155–160.
[11] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, pp. 1550–1560, Oct. 1990.
[12] G. V. Puskorius, L. A. Feldkamp, and L. I. Davis, Jr., "Dynamic neural network methods applied to on-vehicle idle speed control," *Proc. IEEE*, vol. 84, pp. 1407–1420, Oct. 1996.
[13] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems containing neural networks," *IEEE Trans. Neural Networks*, vol. 1, pp. 4–27, Jan. 1990.
[14] G. V. Puskorius and L. A. Feldkamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks," *IEEE Trans. Neural Networks*, vol. 5, pp. 279–297, Mar. 1994.
[15] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, pp. 490–501, 1990.
[16] S. Singhal and L. Wu, "Training multilayer perceptrons with the extended Kalman algorithm," in *Advances in Neural Information Processing Systems 1*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1989, pp. 133–140.
[17] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
[18] S. Haykin, *Adaptive Filter Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
[19] G. V. Puskorius and L. A. Feldkamp, "Decoupled extended Kalman filter training of feedforward layered networks," in *Proc. Int. Joint Conf. Neural Networks*, Seattle, WA, 1991, vol. I, pp. 771–777.
[20] M. B. Matthews, "Neural network nonlinear adaptive filtering using the extended Kalman filter algorithm," in *Proc. Int. Neural Networks Conf.*, Paris, 1990, vol. I, pp. 115–119.
[21] R. J. Williams, "Training recurrent networks using the extended Kalman filter," in *Proc. Int. Joint Conf. Neural Networks*, Baltimore, 1992, vol. IV, pp. 241–246.

[22] L. A. Feldkamp and G. V. Puskorius, “Training of robust neural controllers,” in *Proc. 33rd IEEE Int. Conf. Decision and Control*, Orlando, FL, 1994, vol. III, pp. 2754–2760.
[23] G. V. Puskorius and L. A. Feldkamp, “Extensions and enhancements of decoupled extended Kalman filter training,” in *Proc. 1997 Int. Conf. Neural Networks*, Houston TX, vol. 3, pp. 1879–1883.
[24] A. H. Sayed and T. Kailath, “A state-space approach to adaptive RLS filtering,” *IEEE Signal Processing Mag.*, vol. 11, pp. 18–60, Mar. 1994.
[25] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Trans. Neural Networks*, vol. 5, pp. 157–166, Mar. 1994.
[26] G. V. Puskorius and L. A. Feldkamp, “Signal processing by dynamic neural networks with application to automotive misfire detection,” in *Proc. World Cong. Neural Networks*, San Diego, 1996, pp. 585–590.
[27] K. A. Marko, J. V. James, T. M. Feldkamp, G. V. Puskorius, L. A. Feldkamp, and D. Prokhorov, “Training recurrent networks for classification: Realization of automotive engine diagnostics,” in *Proc. World Congr. Neural Networks*, San Diego, CA, 1996, pp. 845–850.
[28] R. Tawel, N. Aranki, G. V. Puskorius, K. A. Marko, L. A. Feldkamp, J. V. James, G. Jesion, and T. M. Feldkamp, “Custom VLSI ASIC for automotive applications with recurrent networks,” *Proc. 1998 Int. Joint Conf. Neural Networks*, Anchorage AK, vol. 3, pp. 598–602.

Lee A. Feldkamp (Senior Member, IEEE) received the B.S.E. degree in electrical engineering (1964) and the M.S. (1965) and Ph.D. (1969) degrees in nuclear engineering, all from the University of Michigan, Ann Arbor.

He is currently a Senior Staff Technical Specialist with the Powertrain Control Systems Department of Ford Research Laboratory, Ford Motor Company, Dearborn, MI. From 1969 to 1989 he was a Member of the Physics Department at Ford Research, where his interests included neutron scattering, electron energy-loss spectroscopy, theory of electronic spectroscopies, machine vision for robotics, image processing, x-ray tomography, and the biomechanics of bone. In 1989 he formed a group to study neural networks and pursue promising applications. He has published one book (on vibrations in crystal lattices) and authored more than 90 papers and reports. He also has been granted several patents related to neural networks and fuzzy logic.

Dr. Feldkamp is a member of the Board of Governors of the International Neural Network Society and a Fellow of the American Physical Society. He is a member of the editorial board of *Neural Networks* and an Associate Editor of IEEE TRANSACTIONS ON NEURAL NETWORKS.

Gintaras V. Puskorius (Member, IEEE) received the B.S. degree in engineering physics and the M.S. degree in physics from John Carroll University, Cleveland, OH, in 1980 and 1982, respectively.

He joined the Ford Research Laboratory, Ford Motor Company, Dearborn, MI, in 1982, where he is currently a Senior Technical Specialist with the Business Modeling Research group. His past research activities have included studies of alkali metals, laser scanning devices, machine vision and robotics, and image processing for scanning tunneling microscopy. He was a member of Ford Research Laboratory’s Artificial Neural Systems group for eight years, where he pursued both fundamental studies of neural network training methods as well as application of neural network methods to problems in control, diagnostics, and computer-aided engineering. His current activities involve the application of advanced computational methods to problems in business and finance. His research has resulted in seven patents and the publication of over 50 papers and technical reports.

Mr. Puskorius was a co-recipient (with Lee A. Feldkamp) of the 1995 IEEE TRANSACTIONS ON NEURAL NETWORKS Outstanding Paper Award for “Neurocontrol of Nonlinear Dynamical Systems with Kalman Filter Trained Recurrent Networks.”