1 ENHANCED MULTI-STREAM KALMAN FILTER TRAINING FOR RECURRENT NETWORKS

Lee A. Feldkamp, Danil V. Prokhorov,
Charles F. Eagen, and Fumin Yuan

Ford Research Laboratory
Ford Motor Company, Dearborn, Michigan, USA

lfeldkam@ford.com,dprokhor@ford.com,ceagen@ford.com,fyuan@ford.com

Abstract: We present a framework for the training of time-lagged recurrent networks that has been used for a wide variety of both abstract problems and practical applications. Our method is based on rigorous computation of dynamic derivatives, using various forms of backpropagation through time (BPTT), a second-order weight update scheme that uses the extended Kalman filter, and data delivery mechanics designed for sequential weight updates with broad coverage of the available data. We extend our previous discussions of this framework by discussing various alternative forms of BPTT. In addition, we consider explicitly the issue of dealing with and optimizing network initial states. We discuss the initial state problem from the standpoint of making time-series predictions.

1.1 INTRODUCTION

Extended Kalman filter multi-stream training is a synthesis of training techniques that forms a practical and quite general approach to neural network training. The method is comprised of the following elements: 1) gradient calculation by backpropagation through time; 2) weight updates based on the extended Kalman filter; and 3) data presentation using multi-stream mechanics.

With this synthesis, our group and others have dealt with many different types of problems involving temporal systems. In most cases, such problems have had a mixed character, in which there are one or more primary input variables and other inputs which, taken together, provide context. A typical problem might involve prediction, estimation, classification [1, 2], or control [3, 4]. The systems we have encountered have mostly been driven, rather than autonomous. In the present paper, however, we will also consider autonomous systems, in order to focus on network state initialization, a subject largely ignored in most previous work.

Much of our work has made use of *time-lagged recurrent networks*, usually in the form of *recurrent multilayer perceptrons* (RMLP), which are a natural synthesis of feedforward multilayer perceptrons and single-layer fully recurrent networks. In dealing with autonomous systems, especially those which are sensitive to initial states, it may be advantageous to make use of externally recurrent networks, because the network state variables are not hidden. The methods we describe apply just as well to such networks.

Recurrent networks are often found to be hard to train. In addition to compounding the usual problems encountered in training feedforward networks, including poor local minima, recurrent networks exhibit the *recency effect*: the tendency for recent weight updates to cause a network to forget what it has learned in the past. This tendency also exists in the training of feedforward networks, but in that case easily applied countermeasures exist. For example, one may scramble the order of presentation of input-output pairs or employ 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 is important. Although recurrent networks can be trained with batch weight updates, sequential training methods have practical advantages, because feedback on progress of the training process is provided more often, especially for very large data sets. We also are convinced that the stochastic nature of sequential updates often gives rise to better results.

Since recurrent networks are dynamical systems and have well defined state variables, it seems reasonable that the question of how to initialize these state variables would have received a great deal of critical attention. Remarkably, this does not seem to have been the case. In our own work, for example, by dealing largely with stable systems and suitable procedures (see Section 4) that minimize the effect of the network initial state, we have been able almost entirely to bypass such questions, without compromising the quality of network training. However, in a few applications such initialization has assumed some importance. In particular, if a recurrent network is expected to exhibit good performance immediately after execution begins, its initial state may be critical.

If data are scarce, a proper treatment of network initial states might enable all data to be used to advantage in training. This is in contrast to our standard procedure, in which we defer updates until the startup transient of the network is expected to have dissipated, to avoid distorting the network weights on the basis of errors due more to incorrect network state variables than to incorrect weight values.

An obvious situation in which the initial state is important is a network modeling an autonomous system. A chaotic system is an extreme case, because the initial state affects the entire evolution of the network. Here, of course, an accurate long-term model is out of the question, but even the short-term accuracy of a recurrent network model can depend significantly on the choice of initial state.

Finally, we contend that a method of dealing with initial states should permit networks with hidden states, such as RMLPs or recurrent networks in state-space form with nontrivial output functions, to be used in situations where only externally recurrent networks are presently considered. In time-series prediction, for example, it is usual to construct the state of a recurrent network from the measured series values. Even if this is valid, it may be that the evolution of the actual state of the generating system is more simply described than is the evolution of the system output, especially if the output function is not invertible.

We have organized the remainder of this paper as follows. Section 2 presents briefly the recurrent network architecture in the form of an ordered network and defines its evolution. In Section 3, we begin discussion of the training method by describing the calculation of the gradients required for network weight training, presenting two variations of BPTT. Section 4 presents the EKF multi-stream method. Section 4.1 describes how gradients are used in a second-order weight update procedure, based on the extended Kalman filter, while Section 4.2 describes multi-stream training. Section 5 describes a typical modeling application, based on experimental data. Section 6 illustrates the training of both network weights and initial states, using a simple three-state-variable system. We point out how derivatives required for optimizing network initial states arise naturally in BPTT and describe our proposed mechanics for initial state training. In Section 7 we continue discussion of network initial states, demonstrating the extent to which a network's modeling performance can depend on how it is initialized. We present a homogeneous multi-stream training process for a combined network that handles both state initialization and evolution. In Section 8, we illustrate the training process in the context of extended prediction of a time series with several state variables. Section 9 extends our discussion by considering application-specific structured networks. We illustrate the approach by applying it to the time-series competition problem. Finally, in the last section we summarize and make some concluding remarks.

1.2 NETWORK ARCHITECTURE AND EXECUTION

For compactness of presentation, we use the *ordered network* [5] formalism to define both the architecture and execution of recurrent networks. This formalism can describe the basic RMLP form, which consists of one or more layers of computational nodes, as in a standard feedforward network or MLP. In each layer we either have full recurrence (every node is connected through unit time delays to every node) or no recurrence. The ordered network description can also handle sparse connection patterns, as illustrated in Figure 1.1, and is well suited to networks with tapped delay lines, whether internal or external.

The forward equations for an ordered network with `n_in` inputs and `n_out` outputs may be expressed very compactly in a pseudocode format. Let the network consist of `n_nodes` nodes, of which `n_in` serve as receptors for the external inputs. The bias input, which we denote formally as node 0, is not included in the node count. The bias input is set to the constant 1.0. The array **I** contains a list of the input nodes; e.g., I_j is the number of the node that corresponds to the `j`th input, in_j. Similarly, a list of the nodes that correspond to network outputs out_p is contained in the array **O**. We allow network outputs and targets to be advanced or delayed with respect to node outputs by assigning a phase τ_p to each output. For example, if we wish to associate the network output p with the output of some system five steps in the future, we would have $\tau_p = 5$. Node i receives input from `n_con(i)` other nodes and has activation function $f_i(\cdot)$; `n_con(i)` is zero if node `i` is among the nodes listed in the input array **I**. The array **c** specifies connections between nodes; $\mathrm{c}_{i,j}$ is the node number for the `j`th input of node `i`. Inputs to a given node may originate at the current or past time steps, as specified by delays contained in the array **d**, and through weights for time step t contained in the array $\mathbf{W}(t)$.

Most commonly, we take the activation function $f_i(\cdot)$ to be either linear or a bipolar sigmoid, though we also make use of other functions, such as products and sinusoids, for special purposes.

Prior to beginning network operation, all appropriate memory is initialized. Normally, such memory will be set to zero. In some cases, as we discuss later, memory that corresponds to the network initial state may be set to specified values.

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 **d**, and `h` is the truncation depth of the backpropagation through time gradient calculation described in the following section.

```
for i = 1 to n_nodes {
for i_t = t-h-dmax to t-1 {
        $\mathbf{W}(i_t) = \mathbf{W}(i_t+1)$        (1.1)
        $y_i(i_t) = y_i(i_t+1)$        (1.2)
} }
```

The actual network execution is expressed as

```
for i = 1 to n_in {
        $y_{\mathrm{I_i}}(t) = \mathrm{in}_i(t)$        (1.3)
}
for i = 1 to n_nodes {
```

```
if n_con(i) > 0 {
```

$$y_i(t) = f_i\Big(\sum_{j=1}^{\mathrm{n\_con(i)}} W_{i,j}(t) y_{c_{i,j}}(t - d_{i,j})\Big) \qquad (1.4)$$

```
}}
for p = 1 to n_out {
```

$$\mathrm{out}_p(t + \tau_p) = y_{\mathrm{O_p}}(t) \qquad (1.5)$$

```
}
```

Note that the expressions for network execution do not explicitly invoke the concept of *layers*; a layered structure, when desired, 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. Alternatively, an explicit representation of the states that arise from delay lines may be realized by defining nodes that correspond to delayed values. The choice between these representations may be made on the basis of convenience.

1.3 GRADIENT CALCULATION

After the forward propagation at time step t, we compute gradients in preparation for the weight update step. We make use of various forms of truncated backpropagation through time (BPTT(h)) [5, 6]. With the truncation depth h suitably chosen, this method produces accurate derivatives with greatly reduced complexity and computational effort as compared to forward methods such as real-time recurrent learning (RTRL). In the limit $h = 0$, BPTT reduces to ordinary static backpropagation.

We describe here the mechanics of two variations of BPTT(h). The first of these, which we term "traditional BPTT," produces derivatives that are directly comparable to those of RTRL, producing total or *ordered* derivatives of the current network outputs (or errors) with respect to its weights. The other variation, "aggregate BPTT," produces total derivatives of the sum of squared errors over a time horizon equal to the truncation depth.

We use the Werbos notation in which $\overset{q}{\mathrm{F\_}x}$ denotes an ordered derivative of some quantity q 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 $\overset{q}{\mathrm{F\_}b} += \frac{\partial g}{\partial b}\overset{q}{\mathrm{F\_}a}$ and $\overset{q}{\mathrm{F\_}c} += \frac{\partial g}{\partial c}\overset{q}{\mathrm{F\_}a}$. We use the C-language notation "$+=$" to indicate that the quantity on the right hand side is added to the previous 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

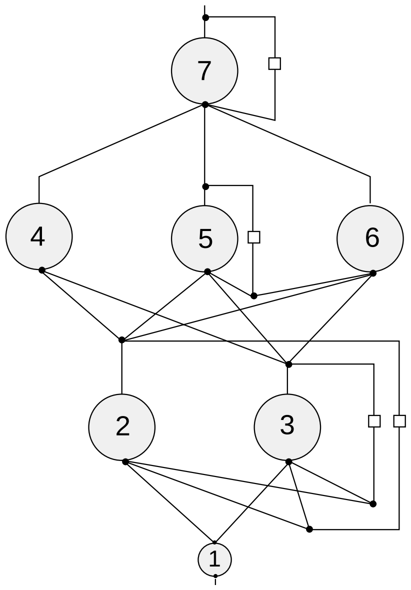

Figure 1.1 Schematic illustration of a sparsely connected RMLP. If both recurrent layers were fully connected, the network would be denoted as 1-2R-3R-1RL. The first hidden layer is fully recurrent, the second hidden layer is partially recurrent, and the output node is recurrent. Nodes 2–6 are bipolar sigmoids, and the recurrent output node is linear. The small boxes denote unit time delays. Bias connections are present, as indicated in Table 1.1, but are not shown.

for visualizations such as unfolding in time or signal-flow graphs.

1.3.1 Traditional BPTT

In traditional BPTT, $\text{F\_}^{p}x$ denotes the ordered derivative of network output node p with respect to x. (It is convenient for application of the Kalman update formalism to focus on derivatives of outputs, rather than derivatives of errors.)

```
for p = 1 to n_out {
for i = 1 to n_nodes {
for $i_t$ = t to t-h-1 {
```

$$\text{F\_}^{p}y_i(i_t) = 0 \tag{1.6}$$

```
}} /* end i and $i_t$ loops */
```

$$\text{F\_}^{p}y_{\mathrm{O_p}}(t) = 1 \tag{1.7}$$

Table 1.1 Connection, delay, and activation function table for the ordered network of Figure 1.1. The jth connection to node i is given by $c_{i,j}$; the corresponding delay is given by $d_{i,j}$. Entries for index i are absent for nodes 0 and 1, since neither receives input from any other node. Note that recurrent connections have unit delays. The elements of the input and output arrays are: $I_1 = 1$ and $O_1 = 7$.

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

$$\xi_p(t) = \mathrm{tgt}_p(t+\tau_p) - \mathrm{out}_p(t+\tau_p) \tag{1.8}$$

```
for i_h = 0 to h {
```

$$i_1 = t - i_h \tag{1.9}$$

```
for i = n_nodes to 1 {
if n_con(i) > 0 {
for k = n_con(i) to 1 {
```

$$j = c_{i,k} \tag{1.10}$$

$$i_2 = i_1 - d_{i,k} \tag{1.11}$$

$$\mathrm{F\_}^{p}y_j(i_2) \quad += \quad \gamma^{d_{i,k}}\mathrm{F\_}^{p}y_i(i_1)W_{i,k}(i_1)f_i'(y_i(i_1)) \tag{1.12}$$
$$\mathrm{F\_}^{p}W_{i,k} \quad += \quad y_j(i_2)\mathrm{F\_}^{p}y_i(i_1)f_i'(y_i(i_1)) \tag{1.13}$$

```
} /* 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 $\frac{\partial y_{\mathrm{O}_p}(t)}{\partial y_{\mathrm{O}_p}(t)} = 1$. The error $\xi_p(t)$ computed in (8) is used in the weight update described in Section 4. The actual backpropagation occurs in expressions (12) and (13), which come 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. The desired value of network output $\mathrm{out}_p(t+\tau_p) = y_{\mathrm{O}_p}(t)$ is denoted as $\mathrm{tgt}_p(t+\tau_p)$.

1.3.2 Aggregate BPTT

In aggregate BPTT, $\mathrm{F\_}^{p}x$ denotes an ordered derivative of $\hat{y}'(t) = -\xi_{p,tot}(t) = (\sum_{i_{\mathrm{h}}=0}^{h} \gamma^{i_{\mathrm{h}}}\,(\xi_p(t-h+i_{\mathrm{h}}))^2)^{\frac{1}{2}}$. Thus, we inject a normalized error at each step of the backpropagation through time (see expression (17) below).

```
for p = 1 to n_out {
for i = 1 to n_nodes {
for i_t = t to t-h-1 {
```

$$\mathrm{F\_}^{p}y_i(i_t) \quad = \quad 0 \tag{1.14}$$

```
}} /* end i and i_t loops */
```

$$\xi_p(t) \quad = \quad \mathrm{tgt}_p(t+\tau_p) - \mathrm{out}_p(t+\tau_p) \tag{1.15}$$

```
for i_h = 0 to h {
```

$$i_1 \quad = \quad t - i_h \tag{1.16}$$
$$\mathrm{F\_}^{p}y_{\mathrm{O_p}}(t) \quad += \quad \xi_p(i_1)/\xi_{p,tot}(t) \tag{1.17}$$

```
for i = n_nodes to 1 {
if n_con(i) > 0 {
for k = n_con(i) to 1 {
```

$$j \quad = \quad c_{i,k} \tag{1.18}$$
$$i_2 \quad = \quad i_1 - d_{i,k} \tag{1.19}$$
$$\mathrm{F\_}^{p}y_j(i_2) \quad += \quad \gamma^{d_{i,k}}\mathrm{F\_}^{p}y_i(i_1)W_{i,k}(i_1)f_i'(y_i(i_1)) \tag{1.20}$$
$$\mathrm{F\_}^{p}W_{i,k} \quad += \quad y_j(i_2)\mathrm{F\_}^{p}y_i(i_1)f_i'(y_i(i_1)) \tag{1.21}$$

```
} /* end k loop */
}
} /* end i loop */
} /* end i_h loop */
} /* end p loop */
```

Another variation of BPTT differs from aggregate BPTT only in the respect that backpropagation to the weights, as in (21), is performed only for $i_t = t-h$. This variation produces derivatives that are equivalent to those of the "temporal backpropagation" scheme of Wan and correspond to the derivatives that are sought by using derivative adaptive critics. As some issues regarding the use of such derivatives in conjunction with Kalman updates remain to be resolved, we do not consider this particular variation here.

1.4 EKF MULTI-STREAM TRAINING

1.4.1 The Kalman Recursion

We have made extensive use of training that employs weight updates based on the extended Kalman filter method first proposed by Singhal and Wu [7]. In most of our work, we have made use of a decoupled version of the EKF method [3, 8], 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 $\mathbf{W}$ 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 $\mathbf{w}_i$. The corresponding derivatives $\mathrm{F\_}\overset{p}{\mathbf{w}}_i$ are placed in `n_out` columns of $\mathbf{H}_i$.

To minimize a cost function $E = \sum_t \frac{1}{2}\boldsymbol{\xi}(t)^T\mathbf{S}(t)\boldsymbol{\xi}(t)$, where $\mathbf{S}(t)$ is a nonnegative definite weighting matrix and $\boldsymbol{\xi}(t)$ is the vector of errors at time step t, the recursion equations for group i are as follows [4]:

$$\mathbf{A}^*(t) = \left[\frac{1}{\eta(t)}\mathbf{I} + \sum_{j=1}^{g}\mathbf{H}_j^*(t)^T\mathbf{P}_j(t)\mathbf{H}_j^*(t)\right]^{-1}, \tag{1.22}$$

$$\mathbf{K}_i^*(t) = \mathbf{P}_i(t)\mathbf{H}_i^*(t)\mathbf{A}^*(t)\,, \tag{1.23}$$

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \mathbf{K}_i^*(t)\boldsymbol{\xi}^*(t)\,, \tag{1.24}$$

$$\mathbf{P}_i(t+1) = \mathbf{P}_i(t) - \mathbf{K}_i^*(t)\mathbf{H}_i^*(t)^T\mathbf{P}_i(t) + \mathbf{Q}_i(t)\,. \tag{1.25}$$

In these equations, the weighting matrix $\mathbf{S}(t)$ is distributed into both the derivative matrices and the error vector: $\mathbf{H}_i^*(t) = \mathbf{H}_i(t)\mathbf{S}(t)^{\frac{1}{2}}$ and $\boldsymbol{\xi}^*(t) = \mathbf{S}(t)^{\frac{1}{2}}\boldsymbol{\xi}(t)$. If traditional BPTT has been used, the error vector $\boldsymbol{\xi}(t)$ has elements $\xi_p(t)$; in the case of aggregate BPTT, we use $\xi_{p,tot}(t)$ instead. See [9] for a discussion of Kalman training with various error functions.

We concatenate the matrices $\mathbf{H}_i^*(t)$ to form a global scaled derivative matrix $\mathbf{H}^*(t)$. We compute a common global scaling matrix $\mathbf{A}^*(t)$ with contributions from all g weight groups, through the scaled derivative matrices $\mathbf{H}_j^*(t)$, and from all of the decoupled approximate error covariance matrices, $\mathbf{P}_j(t)$. A user-specified learning rate, $\eta(t)$, appears in this common matrix. For each weight group i, the Kalman gain matrix $\mathbf{K}_i^*(t)$ is computed and is then used in updating the values of the group's weight vector $\mathbf{w}_i(t)$ and in updating the group's approximate error covariance matrix $\mathbf{P}_i(t)$. Each approximate error covariance update is augmented with the addition of a scaled identity matrix, $\mathbf{Q}_i(t)$, that represents the effects of artificial process noise.

The EKF recursion is typically initialized by setting the approximate error covariance matrices to scaled identity matrices. We usually choose 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 process noise parameter, e.g., $\mathbf{Q}_i(0) = 10^{-2}\mathbf{I}$. We have previously demonstrated that artificial process noise accelerates training, helps to avoid poor local minima during training, and tends to keep approximate error covariance matrices nonnegative definite, as required [8]. As training progresses, we generally decrease $\mathbf{Q}_i$. The training dynamics depend on which form of BPTT is used for derivative calculation (the above stated values are based largely on our experience with traditional BPTT).

1.4.2 Multi-Stream Training

The standard recurrent network training problem involves training on sequential input-output pairs. If the sequence is fairly homogeneous, then one or more sequential passes through the data will probably produce good results. In many training problems, however, 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, the tendency always exists for the network weights to be adapted to the currently presented training data at the expense of performance on previous data. This *recency effect* is analogous to the difficulty that may arise in training feedforward networks if training data are presented always in the same order.

The multi-stream procedure largely circumvents the recency effect by combining features of both scrambling and batch updates. Like full batch methods, multi-stream training [10] 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 than does a batch update method.

Multi-stream training can also be applied to unconventional training problems, such as training robust controllers [11], training dynamic networks to exhibit behavior usually attributed to adaptive systems [12, 13], and encouraging stability of recurrent networks used as models [10, 14].

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 multi-stream 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.

In the absence of prior information with which to initialize the recurrent network, we initialize the outputs of all state nodes to zero at the start of each stream. Correspondingly, the network is executed for a number of steps N_t, the *trajectory length*, but updates are suspended for N_p time steps, called the *priming length*, at the beginning of each stream. Hence $N_t - N_p$ updates are performed in each training cycle. This priming scheme is based on the fact that the outputs of a stable network, after a suitable number of time steps, will be essentially independent of its initialization. A related quantity, N_e, denotes the number of steps for which injection of error as in expression (17) is suspended; in most applications $N_p = N_e$.

Generally speaking, in return for somewhat increased computational overhead, 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 is reasonable not to make the trajectory length N_t too large.

An important consideration is how to carry out a consistent EKF update procedure. To do this, we treat the training problem as that of a single shared-weight network in which the number of original outputs is multiplied by the number of streams. A practical limit on the number of streams may be set by the need to invert a matrix whose dimension is $N_s \times \texttt{n\_out}$.

In single-stream EKF training, we place derivatives of network outputs with respect to network weights in the matrix $\mathbf{H}$ constructed from `n_out` column vectors, each of dimension equal to the number of trainable weights, N_w. In multi-stream training, the number of columns is correspondingly increased to

$N_s \times$ `n_out`. Similarly, the vector of errors $\boldsymbol{\xi}$ has $N_s \times$ `n_out` elements. Apart from these augmentations of $\mathbf{H}$ and $\boldsymbol{\xi}$, the form of the Kalman recursion is unchanged. The Kalman recursion produces weight updates which are not a simple average of the weight updates that would be computed separately for each output or stream. A plausibility argument for the efficacy of these updates is presented in [14].

1.5 A MODELING EXAMPLE

In this section we describe a typical use of multi-stream EKF to train a recurrent network to model a physical system. To avoid proprietary issues, we shall not discuss the context in which the data used here arise, except to say that they were obtained from a physical system as part of an existing development process. The object of the training exercise was to determine the extent to which a set of easily acquired signals could be used to synthesize a close replica of the output of a sensor that is too difficult or costly for routine use. This is the classic statement of a "virtual sensor" problem.

In this example, we have six variables available as time sequences for network inputs; each of these represents a sampled signal and reflects minor effects from noise and other measurement irregularities. There is a single target output sequence, representing measurements from a sensor installed at considerable cost and effort. The present system is driven by external actions according to several different types of trajectories. The external drivers are not explicitly available, but their effect is evident in the network input sequences. As is frequently the case in such applications, there was no *a priori* guarantee that the information contained in the input sequences would be sufficient to model the target output. The different types of external excitation give rise to a rather large dynamic range for the target output. Indeed, the best approach in an engineering sense might be to divide the problem into several behavioral modes and to construct a separate model for each mode. Here, however, we assert that a single model is feasible. Training by standard methods, e.g., by presenting the various files one-by-one, would almost surely suffer the consequences of the recency effect, especially as many files contain long segments in which both inputs and target output change quite slowly. The kernel idea of multi-stream training is that, by basing each weight update on several different data segments, we can capture the required range of behavior in the trained recurrent network.

The available data consists of 84 files, together representing more than 200,000 time steps. Each file consists of a natural trajectory of system operation, and the various trajectories represent chosen external patterns of excitation. Because of the nature of this virtual sensor application, it is permissible to delay the network output relative to the input. Here we chose $\tau_p = -25$ in (5). That is, the target at time step t is the system output at time step $t - 25$.

On the basis of initial explorations, we chose to use an RMLP of architecture 6-7R-5R-3R-1. Training was carried out using 25 streams, traditional BPTT with truncation depth $h = 29$, a priming length $N_p = N_e = 20$, a trajectory

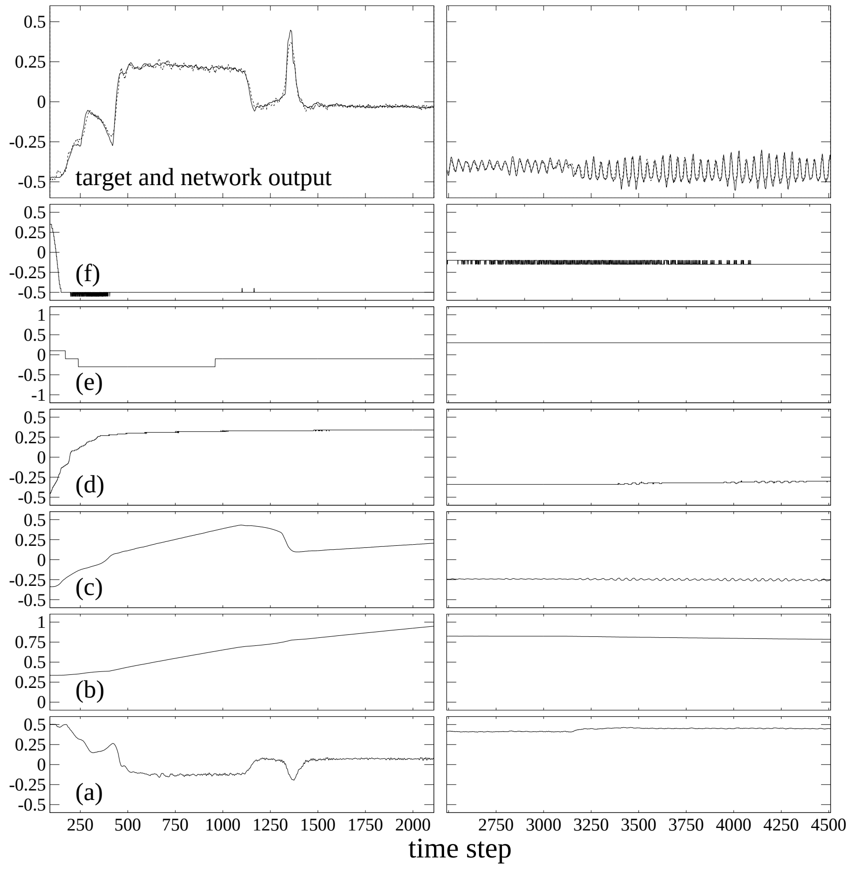

Figure 1.2 Plots of the input variables, the target output variable, and the network output for portions of two different trajectories from the modeling problem discussed in the text. Panels labeled a-f show network inputs. In the top panels, the target output is the solid line, while the network output is dashed.

length $N_t = 200$, and global EKF. By the end of the training process, each instance of the training data had been processed about four times.

Figure 1.2 shows network inputs, target outputs, and network outputs for 2000-step segments of two files; all variables reflect the scaling used in the training process. Note how different are the target output sequences shown in the top panels. The RMS errors for these sequences are representative of the error averaged over the entire training set. Overall, we find remarkable the fact that the network manages to capture quite accurately both the absolute value of the target variable and details such as low-amplitude oscillation. Though the target most closely reflects input (a), the relationship is clearly nonlinear.

1.6 ALTERNATING TRAINING OF WEIGHTS AND INITIAL STATES

In this section, we consider initial state training as it might arise in the context of modeling a stable dynamical system. For such systems, our approach heretofore has been to defer making updates at the start of each trajectory until initial-state transients of both the modeled system and the modeling network have sufficiently diminished. However, for some problem statements, including the one that follows, this scheme is not completely adequate because the effects of initial state cannot be ignored.

Suppose we have the linear dynamical system represented by the sparsely connected 1-3RL-1L network specified in Table 1.2. This is an abstract version of a potentially important practical modeling problem. For simplicity, let the system be driven by an input sequence whose values are randomly chosen from the range [0,1]. We assume that only disjoint input-output segments of length 20 are available, and assume that we know the correct network structure, but not the weights. However, without loss of generality we can choose the weights of node 5 to be fixed at unity. The state of the system (i.e., the outputs of the 3 recurrent nodes) is unknown at the beginning of each data segment.

Table 1.2 Connection, delay, and activation function table for a 1-3RL-1L network. The weights used to generate data for the example are shown. Note that no bias connections are present. The elements of the input and output node arrays are: $I_1 = 1$ and $O_1 = 5$.

i	j	$c_{i,j}$	$d_{i,j}$	$W_{i,j}$	activation
2	1	1	0	.025	linear
	2	2	1	.95	
3	1	1	0	.2	linear
	2	3	1	.2	
4	1	1	0	.2375	linear
	2	4	1	.05	
5	1	2	0	1	linear
	2	3	0	1	
	3	4	0	1	

To apply multi-stream training to this problem, we deviate slightly from the procedure as described above, by locking each stream to a particular data segment. However, a difficulty arises because the available data sequences are too short to support a priming length that will completely eliminate the effect of the unknown initial state. On the other hand, setting the priming length to zero forces the training process to compensate for unknown initial states by distorting the network weights. For example, training with the priming length set to zero produced an RMS error of 0.159, averaged over the 10 streams that correspond to the 10 sets of 20 input-output pairs. Training with the priming length set to 5 reduced the RMS error to 0.087. In both cases, the error was large at the beginning of each segment and decreased quickly, as shown in Figure 1.3.

The ten-stream training process was begun using a priming length of 5, with each data point being processed 400 times. Then initial state training was performed, with the network weights fixed, starting with values of 0.5 for each of the 3 recurrent nodes. Reflecting the physical origin of the model from which the network was specified, both the network weights and its initial states were constrained to be nonnegative, using the scheme described in [9]. The maximum time horizon was chosen to be $m = 20$. Aggregate BPTT was carried out at $t = 20$ only. (This is accomplished with $N_p = 19$, $N_e = 0$, and $h = 20$.) Derivatives of the sum of squared errors for steps 1 through 20 with respect to node outputs at step 0 are computed in this way. Using such gradients, an EKF update of $y_2(0)$, $y_3(0)$, and $y_4(0)$ can be made. This process was repeated several times for each segment separately. The RMS error was reduced from 0.087 to 0.033, largely because of significant improvement at the beginning of each segment, as shown in Figure 1.3. Finally, network weight training was repeated with the learned initial states instantiated. This step reduced the error to 0.027. Additional iterations reduce the error only slightly.

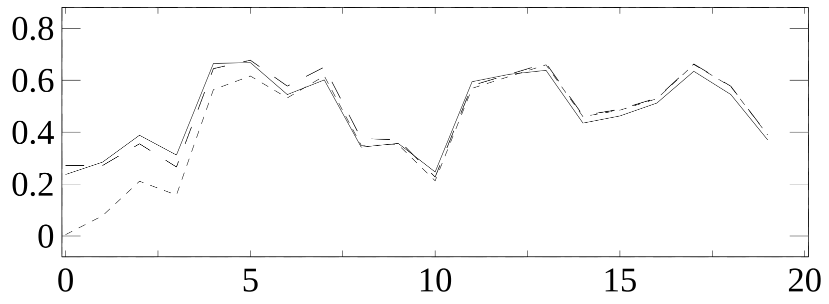

Figure 1.3 The solid curve is the output of the model. The short-dashed line is the output of a network trained with initial states set to zero. The long-dashed line is the output of the same network after its initial states have been trained.

In this section, we have illustrated how initial state training can help in further reducing transient errors for problems with data segments too short to be handled effectively by regular priming. Here training weights was followed by initial state training. While such an alternation is convenient to use for stable systems, it is not the only possibility to enhance the network performance. Another approach is introduced and discussed in the subsequent sections. A key feature of this more general approach is that it can be applied in applications such as prediction, where information upon which to base a retrospective state adjustment is not available.

1.7 TOWARD MORE COORDINATED TRAINING OF WEIGHTS AND STATES–A MOTIVATING EXAMPLE

In some cases, the initial state of a network may influence its evolution for many time steps. This effect is extreme when predicting or modeling a chaotic time series. In this section we show that, somewhat remarkably, there exists a systematic method by which the state of a network predictor may be initialized to significantly enhance its performance.

Consider the following time series, a logistic map.

$$y(t) \quad = \quad 3.9y(t-1)(1-y(t-1)) \tag{1.26}$$

Suppose we generate a long sequence of values of y from (26) and assemble from this sequence a one-input, one-output feedforward network training problem. Of course, this problem is very simple, and the input-output function can be trained to essentially arbitrary accuracy. For present purposes, we employ a relatively small network, say 1-5-1L, with five bipolar sigmoid hidden nodes and a linear output node. After training with EKF for 1500 cycles through a data set of 1000 instances we reduce the RMS error to 0.008. Now let this network be employed as an iterated predictor of the time series. Despite the network's having captured closely the quadratic relationship between input and output, accumulation of small errors leads fairly quickly to rather large prediction errors. This behavior is well known in this and many other problems. Usually the difficulty is attributed to the network having been trained only for one-step prediction, rather than for the iterated prediction for which it is being tested. While this analysis has merit (especially in problems in which training for one-step prediction tends to lead to simple extrapolation), it tends to obscure an important point. In the present problem, a significant improvement can be observed *without changing the network*, merely by adjusting the network's initial state. Since, as usual, iterated prediction is begun by initializing the network based on values of the series (here just one point is required), it seems counterintuitive that a different value could lead to better prediction, but that is indeed the case.

We first proceed in a fashion similar to that of the previous section. Denoting a given starting point of the series as point 0, we adjust the network initial state to minimize errors over a prediction horizon consisting of points 1 through m, while holding the prediction network fixed. We proceed in stages, beginning with the first two points and then training over progressively larger numbers of points until all m have been included. The gradients are obtained by aggregate BPTT. When this process has been carried out over the specified horizon, we find that agreement between the network output and the time series has improved considerably. We emphasize that this improvement is attained without altering the network itself. Though the network output here can not be properly termed a prediction (the initial state is trained on the target series values), we presently will show that this reservation may be overcome.

We can repeat this process for all horizons of length m. For a series of length n, we can make $n-m$ initial state adjustments. In Figure 1.4 we illustrate the change in initial state variable as a function of the nominal state variable, based on 3000 initial state adjustments for the logistic map and the network used above. The average RMS error over horizon $m = 15$ for nominal initial state variables is 0.317; the corresponding error for adjusted initial states is 0.044, clearly a large improvement. Using a simple interpolation, we can then compute adjustments to initial states for portions of the series not included in the training. Statistics for a large number of such continuations confirm a significant improvement in RMS errors: 0.318 (nominal initial state) vs. 0.103 (adjusted initial state). The improvement is not as large as observed in training, presumably because of extreme sensitivity to initial state errors that arise from the approximation inherent in the mapping process.

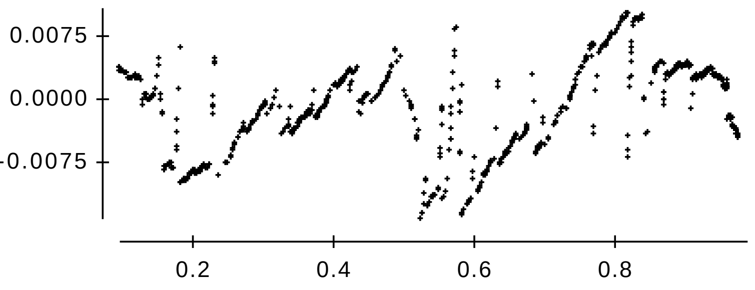

Figure 1.4 A plot of the network initial state adjustment learned by carrying out the procedure outlined in the text.

The fact that the mapping of Figure 1.4 has some generalization value (i.e., predictions are improved for novel portions of the series) suggests a more homogeneous approach, in which we attempt to train directly a subnetwork to perform the initial state adjustment. The idea is to convert available information, here a single value of the time series, into network initial states. We term the subnetwork that performs such an operation the *initiator* and term the subnetwork that is responsible for state evolution the *evolver*. Figure 1.5 illustrates the combined initiator-evolver network, where parameter α takes on values 0 or 1 and is provided to reflect the fact that the initiator feeds the evolver for a limited number of time steps at the start of network operation. Since in the present case the state is one-dimensional, α is unity for the first time step and zero thereafter.

We used such a structure for the logistic map (26). The initiator was chosen to be a 1-10-1L MLP; the evolver had the same architecture as used above, 1-5-1L. We carried out training for iterated prediction with GEKF to horizon $m = 17$, attained sequentially starting from $m = 1$ with unit increments. We obtained an RMS training error of 0.0026. More significantly, the combined network, tested on novel data, produced an error of 0.0054. Carrying out the same training procedure for the evolver alone (i.e., nominal initial states are used), the results are substantially inferior: training error of 0.0090, testing error 0.0125. Figure 1.6 compares sequences for the two cases.

To conclude this section, we note that similar performance can be obtained from an initiator with fewer nodes. The initiator can also be structured with a fixed jump connection from input to output, so that the trainable part of the network learns only the state adjustment.

In this section, we have introduced the combined initiator-evolver network for problems where the effect of initial states on system dynamics can be persistent. We have transformed seemingly separate training of initial states into equivalent training of weights of the initiator. Because the initiator requires only available information as input, it can be used for prediction. It allows us not only to gain substantially in network performance but also to achieve good generalization.

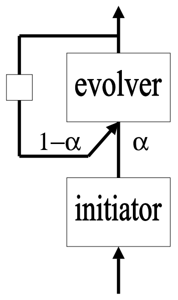

Figure 1.5 Schematic illustration of a combined initiator-evolver network. The initiator provides values for the initial state variables of the evolver. The parameter α is unity for a specified number of time steps and zero thereafter.

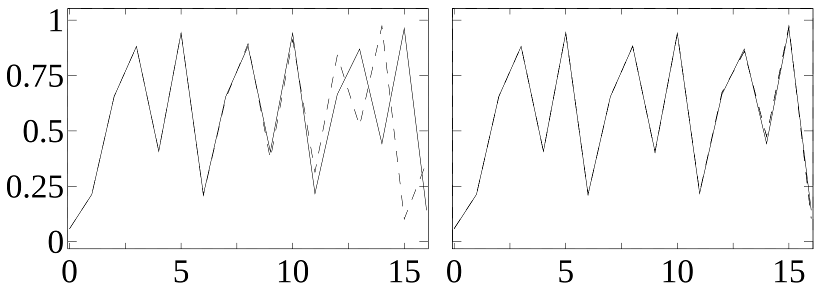

Figure 1.6 Iterated predictions of a portion of a logistic map not used in training. In the left plot, the solid line represents 17 steps of the actual series, while the dashed line is the prediction of a trained evolver network. The right plot shows results for the combined initiator-evolver network. Depending on the starting point, the improvement is not always this evident.

1.8 THE MACKEY-GLASS SERIES

The Mackey-Glass (MG) series has frequently been used to illustrate time series prediction methods and may be more representative of practical series than is the logistic map. Here we first apply the initiator-evolver approach to the simpler of the two commonly employed versions ($\tau = 17$).

The evolver subnetwork has the architecture specified by Table 1.3. This network contains 5 hidden nodes and a linear output node. We begin by training a network for direct (not iterated) six-step prediction. To do this, we hold the weights W_{21} and W_{22} fixed at 1 and 0, respectively. In effect, the inputs are obtained from 5 taps on a delay line with tap spacing 6. (Alternatively, we could have prepared explicitly a file containing appropriate input vectors and used a feedforward network with architecture 5-5-1L.) We set the trajectory length to $N_t = 31$ and the priming length $N_p = 30$, so that weight updates occur only when the delay line has been filled. The training set consisted of 1880 series values. The RMS error for the resulting one-step prediction network is 0.0083.

Table 1.3 Network architecture used for the Mackey-Glass series. The input stream consists of consecutive values of the time series. The network output is ignored for 30 time steps, during which series values populate the buffer that holds present and past values of node outputs. If $\alpha = 1$, the network operates as a direct 6-step predictor, while for $\alpha = 0$ it performs pure iterated prediction. Values for the weights of nodes 3–8, not shown here, are initialized randomly to small values before training begins. Here $\tau_p = 1$.

i	j	$c_{i,j}$	$d_{i,j}$	$W_{i,j}$	activation
2	1	1	0	α	linear
	2	8	6	$1 - \alpha$	
3-7	1	0	0		bipolar sigmoid
	2	2	0		
	3	2	6		
	4	2	12		
	5	2	18		
	6	2	24		
8	1	0	0		linear
	2	3	0		
	3	4	0		
	4	5	0		
	5	6	0		
	6	7	0		

By switching α from 1 to 0 at time step 31, we can use this network to perform iterated prediction. During the first 30 steps, series values set the network initial state variables. At step 31, the network begins to make predictions, which then are fed back as network input. For 120-step predictions, we obtain an RMS error of 0.029; this represents the average of iterated predictions from 1880 different starting points. On novel data, the error is increased only slightly, to 0.030. When the evolver is explicitly trained for iterated prediction, its performance on the training set is improved to 0.019.

In contrast to the logistic map, the network initial state here is multi-dimensional. Hence, various ways of employing an initiator are available. We consider here the simplest possibility, which can be regarded as a serial adjustment of the sequence of series values to form the network initial state. We place a single-input, single-output feedforward network between nodes 1 and 2 of the network of Table 1.3. This initiator is a 1-10-1L MLP, augmented with a fixed jump connection between its input and output, so that the MLP learns adjustments to the series values. We start with the adjustable weights of the initiator randomly initialized, and the evolver weights set to those of the direct 6-step predictor. Ten-stream training for iterated prediction was carried out with the trajectory length increased in stages. The BPTT truncation depth was set to $N_t - 1$, i.e., large enough to capture the sensitivity of evolver output to outputs of the initiator network at each of the first 30 time steps of each trajectory, during which initial state of the evolver is being formed. Coordinated training of the initiator and evolver was accomplished using GEKF updates. At each stage, the network was tested on all possible trajectories with a 120-step horizon ($N_t = 150$, $N_t - N_p = 120$). The final RMS training error is 0.012. The same value was found for the novel data, suggesting that the combined network had not only learned the essence of the evolution of the series but also had learned how to form an initial state to optimize its performance. Further, the additional parameters required for the initiator have not compromised the network's generalization.

We repeated the process for the more difficult version of the series, $\tau = 30$. In coordinated training of the initiator and evolver, we achieved an RMS error of 0.035, while testing on novel data yielded 0.040. Typical iterated predictions for testing trajectories are shown in Figure 1.7.

Summarizing this section, we have verified viability of a coordinated training of the initiator and the evolver for a chaotic system with multi-dimensional initial state.

1.9 USE OF A STRUCTURED EVOLVER

In this section, we would like to discuss a promising generalization of the approach illustrated above, motivated by the time-series competition problem, for which a straightforward application of iterated prediction training seemed to be inadequate for extended predictions.

The competition series may be described as having oscillating behavior with slowly varying amplitude and frequency, along with occasional jumps of the level about which the oscillations are centered. Since it is not entirely trivial merely to train a network to exhibit sinusoidal oscillations, it is appealing to consider specialized node activation functions. Here we used a two-block struc-

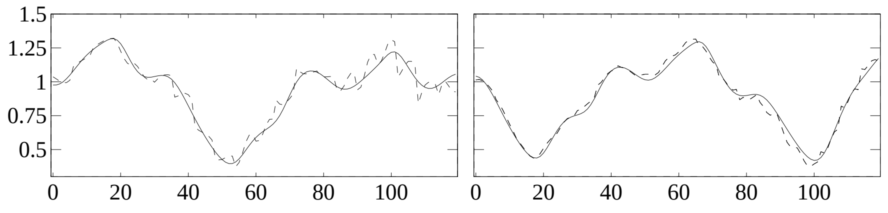

Figure 1.7 Iterated prediction (horizon of 120 time steps) for the Mackey-Glass series, $\tau = 30$, using the initiator-evolver network. The 30 time steps of the series that precede the trajectories shown here were processed by the initiator to provide an initial state for the evolver. The actual series is drawn with a solid line; the network output is dashed.

ture, where the first block contains sinusoidal nodes and produces oscillatory behavior intrinsically. This block is driven by a second block, upon which the training effort is concentrated. Justification for using such a structured evolver is based on potentially increased efficacy of training, not on increased representational capability. In particular, we argue that training a recurrent network to oscillate with slowly varying amplitude, frequency and offset to follow a target series is harder than training a network to evolve these controlling variables themselves.

As noted above, we encountered difficulty in training a standard externally recurrent network, such as that used for the Mackey-Glass series, to provide good iterated predictions beyond, say, 20 or 30 steps. On the other hand, a local fit to the parameters of a single biased sinusoid often produced respectable agreement over the same interval. Further, having produced local-fit parameters for all 30-step horizons supported by the data, we observed a considerable degree of predictability of one set of parameters from preceding sets of parameters, suggesting that successful training of an evolver with these parameters as state variables might be possible.

We chose the following evolver state variables: amplitude $a(t)$, bias (or dc offset) $b(t)$, frequency (more properly, sinusoid argument increase per time step) $\omega(t)$, $s(t)$, and $c(t)$. The variables $s(t)$ and $c(t)$ may be regarded as sine and cosine states that together encode the current phase. The upper block of the evolver is implemented with sine and cosine nodes, product nodes, and linear nodes. The lower block is externally recurrent with an internal tapped delay line. It has as inputs the five one-step-delayed state variables listed above; nine taps, spaced nine steps apart, are devoted to each variable. The kernel of this block is an MLP with structure 45-15-10-3L and outputs $a(t)$, $b(t)$, and $\omega(t)$. All trainable weights of the evolver are associated with this block.

A sine-generating recurrent block carries out the following recursion:

$$\begin{aligned} s(t) &= s(t-1)\cos(\omega(t-1)) + c(t-1)\sin(\omega(t-1)) && (1.27)\\ c(t) &= c(t-1)\cos(\omega(t-1)) - s(t-1)\sin(\omega(t-1)) && (1.28) \end{aligned}$$

To obtain the final output, we multiply $s(t)$ of (27) by the amplitude and add the bias:

$$y_{out}(t) = a(t) * s(t) + b(t) \qquad (1.29)$$

This output is to be compared with the corresponding value of the time series, as in (8).

In the present case, we chose the delay line of the evolver to have length 81, with taps spaced at 9 step intervals. Thus the initial state of the evolver requires 81 values of each of the 5 state variables. This could, in principle, be carried out by an initiator network, based on past values of the time series. At present, however, we take the expedient of using values obtained by locally fitting the series. Hence, the network operates by inputting 81 values of each of the 5 input variables, then taking its input as previous values of the 5 evolver state variables. Apart from having five delay lines instead of one, the mechanics are the same as described above for other time series.

Training was carried out using node-decoupled DEKF rather than GEKF for initial experiments, in view of the large number of adjustable weights. Multi-stream training with 10 streams was employed. The 45-15-10-3L evolver was initialized with values obtained by training for direct 9-step prediction of states a, b, and ω. The trajectory length was increased in stages, from 111 (prediction horizon $111 - 81 = 30$) to 201. The truncation depth h was set to $N_t - 81 - 1$, with a maximum of $h = 120$. The time required for training is substantial, because of the number of weights and the depth of BPTT. At the start of training, the RMS error averaged over all possible 120-step horizons of iterated prediction was 0.264, about the same as the standard deviation of the series itself. After 2000 cycles of training, which represents 182,000 weight updates, the error had been reduced to 0.092. Though this value is not low enough for us to have confidence in a prediction of the series beyond the provided data, the steady improvement observed during training is promising. In addition, it seems quite likely that tailoring of the training procedure to this problem will yield significantly lower error. Figure 1.8 illustrates iterated prediction for three typical trajectories with horizon 120. The sequence on the right is rather well predicted, while the other two suggest that the network is having difficulty capturing details of the change of bias and amplitude. Even so, the evolver seems to be on its way to learning the essence of the dynamical system that underlies the series. We recognize that we have no evidence that the network would generalize to novel portions of the series; further work will clarify this. In addition, we have not explicitly considered the very real possibility that the underlying system is nonstationary.

As mentioned, we have not yet applied the coordinated initiator-evolver training scheme to this problem. It is quite possible that the initial states instantiated on the basis of local fits are limiting the network's performance. If this is indeed the case, then we would expect an improvement by coordinating training of the evolver with that of an initiator that transforms the local-fit-derived network initial states into ones that are more optimal. Naturally, such a training process will be slower due to the increased size of the combined network.

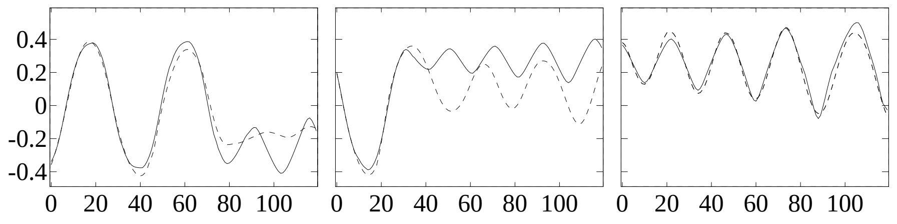

Figure 1.8 Iterated predictions (horizon of 120 time steps) from three randomly chosen starting points of the competition time series. The actual series is drawn with a solid line; the network output is dashed.

1.10 SUMMARY

For efficient training of recurrent networks for problems where initial states are important, we advocate coordinated training of the evolver and the initiator. The initiator receives a vector of delayed values of the time series as its input and outputs a vector which serves as initial states of the evolver. The evolver is responsible for state evolution and trained to match values of the time series. Based on the results of the previous sections, a procedure for coordinated training of the initiator and the evolver can be summarized as follows.

```
Choose maximum time horizon of iterative predictions $m$.
Instantiate input tapped-delay lines of the initiator based on
 series values from steps $\leq 0$.
For $t = 2$ to $m$ in chosen steps
{
Execute the initiator for as many time steps as the number of
unit-step delays in the input taps of the evolver.
Set $h = t - 1$.
Run the evolver forward until time step $t$.
Carry out aggregate BPTT from $t$ to 1 as described in
 Section 3.2 to obtain gradients with respect to weights
 of both the evolver and initiator.
Perform GEKF-based weight updates for both networks.
}
```

Our experience to date suggests that the coordination of weight updates for the initiator and the evolver provided by GEKF, as compared to DEKF, is important.

In summary, we have described the multi-stream EKF method for training neural networks and have articulated an approach to dealing with network initial states. The initiator-evolver method can easily be generalized to applications other than time series prediction. In particular, the proposed structure has already proven useful in a practical application, not presented here, where it was useful to instantiate network initial states on the basis of available information that was not used in network training.

Acknowledgments

We wish to thank our colleague Gintaras Puskorius for his contributions over many years to the methods that form the basis for the work reported here.

References

[1] G. V. Puskorius and L. A. Feldkamp, “Signal processing by dynamic neural networks with application to automotive misfire detection,” in *Proceedings of the World Congress on Neural Networks*, San Diego, 1996, pp. 585–590.

[2] 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 *Proceedings of the World Congress on Neural Networks*, San Diego, 1996, pp. 845–850.

[3] G. V. Puskorius and L. A. Feldkamp, “Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 279–297, 1994.

[4] G. V. Puskorius, L. A. Feldkamp, and L. I. Davis, Jr., “Dynamic neural network methods applied to on-vehicle idle speed control,” *Proceedings of the IEEE*, vol. 84, no. 10, pp. 1407–1420, 1996.

[5] P. J. Werbos, “Backpropagation through time: What it does and how to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[6] 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.

[7] S. Singhal and L. Wu, “Training multilayer perceptrons with the extended Kalman algorithm,” in D. S. Touretzky (ed), *Advances in Neural Information Processing Systems 1*, pp. 133–140. San Mateo, CA: Morgan Kaufmann, 1989.

[8] G. V. Puskorius and L. A. Feldkamp, “Decoupled extended Kalman filter training of feedforward layered networks,” in *Proceedings of the International Joint Conference on Neural Networks*, Seattle, 1991, vol. I, pp. 771–777.

[9] G. V. Puskorius and L. A. Feldkamp, “Extensions and enhancements of decoupled extended Kalman filter training,” in *Proceedings of the 1997 In-*

ternational Conference on Neural Networks, Houston TX, 1997, vol. 3, pp. 1879–1883.

[10] G. V. Puskorius and L. A. Feldkamp, "Multi-stream extended Kalman filter training for static and dynamic neural networks," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, FL, 1997, vol. 3, pp. 2006–2011.

[11] L. A. Feldkamp and G. V. Puskorius, "Training of robust neural controllers," in *Proceedings of the 33rd IEEE International Conference on Decision and Control*, Orlando, 1994, vol. III, pp. 2754–2760.

[12] L. A. Feldkamp, G. V. Puskorius, and P. C. Moore, "Adaptive behavior from fixed weight networks," *Information Sciences*, vol. 98, pp. 217–235, 1997.

[13] L. A. Feldkamp and G. V. Puskorius, "Fixed weight controller for multiple systems," *Proceedings of the 1997 International Conference on Neural Networks*, Houston TX, 1997, vol. 2, pp. 773–778.

[14] L. A. Feldkamp and G. V. Puskorius, "A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering and classification," *Proceedings of the IEEE*, vol. 86 no. 11, pp. 2259–2277, 1998.