Kalman Filtering and Neural Networks, Edited by Simon Haykin

ISBNs: 0-471-36998-5 (Hardback); 0-471-22154-6 (Electronic)

2

PARAMETER-BASED KALMAN FILTER TRAINING: THEORY AND IMPLEMENTATION

Gintaras V. Puskorius and Lee A. Feldkamp

Ford Research Laboratory, Ford Motor Company, Dearborn, Michigan, U.S.A.
(gpuskori@ford.com, lfeldkam@ford.com)

2.1 INTRODUCTION

Although the rediscovery in the mid 1980s of the backpropagation algorithm by Rumelhart, Hinton, and Williams [1] has long been viewed as a landmark event in the history of neural network computing and has led to a sustained resurgence of activity, the relative ineffectiveness of this simple gradient method has motivated many researchers to develop enhanced training procedures. In fact, the neural network literature has been inundated with papers proposing alternative training

methods that are claimed to exhibit superior capabilities in terms of training speed, mapping accuracy, generalization, and overall performance relative to standard backpropagation and related methods.

Amongst the most promising and enduring of enhanced training methods are those whose weight update procedures are based upon second-order derivative information (whereas standard backpropagation exclusively utilizes first-derivative information). A variety of second-order methods began to be developed and appeared in the published neural network literature shortly after the seminal article on backpropagation was published. The vast majority of these methods can be characterized as batch update methods, where a single weight update is based on a matrix of second derivatives that is approximated on the basis of many training patterns. Popular second-order methods have included weight updates based on quasi-Newton, Levenburg–Marquardt, and conjugate gradient techniques. Although these methods have shown promise, they are often plagued by convergence to poor local optima, which can be partially attributed to the lack of a stochastic component in the weight update procedures. Note that, unlike these second-order methods, weight updates using standard backpropagation can either be performed in batch or instance-by-instance mode.

The extended Kalman filter (EKF) forms the basis of a second-order neural network training method that is a practical and effective alternative to the batch-oriented, second-order methods mentioned above. The essence of the recursive EKF procedure is that, during training, in addition to evolving the weights of a network architecture in a sequential (as opposed to batch) fashion, an approximate error covariance matrix that encodes second-order information about the training problem is also maintained and evolved. The global EKF (GEKF) training algorithm was introduced by Singhal and Wu [2] in the late 1980s, and has served as the basis for the development and enhancement of a family of computationally effective neural network training methods that has enabled the application of feedforward and recurrent neural networks to problems in control, signal processing, and pattern recognition.

In their work, Singhal and Wu developed a second-order, sequential training algorithm for static multilayered perceptron networks that was shown to be substantially more effective (orders of magnitude) in terms of number of training epochs than standard backpropagation for a series of pattern classification problems. However, the computational complexity of GEKF scales as the square of the number of weights, due to the development and use of second-order information that correlates every pair of network weights, and was thus found to be impractical for all but

the simplest network architectures, given the state of standard computing hardware in the early 1990s.

In response to the then-intractable computational complexity of GEKF, we developed a family of training procedures, which we named the *decoupled* EKF algorithm [3]. Whereas the GEKF procedure develops and maintains correlations between each pair of network weights, the DEKF family provides an approximation to GEKF by developing and maintaining second-order information only between weights that belong to mutually exclusive groups. We have concentrated on what appear to be some relatively natural groupings; for example, the *node-decoupled* (NDEKF) procedure models only the interactions between weights that provide inputs to the same node. In one limit of a separate group for each network weight, we obtain the *fully decoupled* EKF procedure, which tends to be only slightly more effective than standard backpropagation. In the other extreme of a single group for all weights, DEKF reduces exactly to the GEKF procedure of Singhal and Wu.

In our work, we have successfully applied NDEKF to a wide range of network architectures and classes of training problems. We have demonstrated that NDEKF is extremely effective at training feedforward as well as recurrent network architectures, for problems ranging from pattern classification to the on-line training of neural network controllers for engine idle speed control [4, 5]. We have demonstrated the effective use of dynamic derivatives computed by both forward methods, for example those based on real-time-recurrent learning (RTRL) [6, 7], as well as by truncated backpropagation through time (BPTT(h)) [8] with the parameter-based DEKF methods, and have extended this family of methods to optimize cost functions other than sum of squared errors [9], which we describe below in Sections 2.7.2 and 2.7.3.

Of the various extensions and enhancements of EKF training that we have developed, perhaps the most enabling is one that allows for EKF procedures to perform a single update of a network's weights on the basis of more than a single training instance [10–12]. As mentioned above, EKF algorithms are intrinsically sequential procedures, where, at any given time during training, a network's weight values are updated on the basis of one and only one training instance. When EKF methods or any other sequential procedures are used to train networks with distributed representations, as in the case of multilayered perceptrons and time-lagged recurrent neural networks, there is a tendency for the training procedure to concentrate on the most recently observed training patterns, to the detriment of training patterns that had been observed and processed a long time in the past. This situation, which has been called the *recency*

phenomenon, is particularly troublesome for training of recurrent neural networks and/or neural network controllers, where the temporal order of presentation of data during training must be respected. It is likely that sequential training procedures will perform greedily for these systems, for example by merely changing a network's output bias during training to accommodate a new region of operation. On the other hand, the off-line training of static networks can circumvent difficulties associated with the recency effect by employing a scrambling of the sequence of data presentation during training.

The recency phenomenon can be at least partially mitigated in these circumstances by providing a mechanism that allows for multiple training instances, preferably from different operating regions, to be simultaneously considered for each weight vector update. *Multistream* EKF training is an extension of EKF training methods that allows for multiple training instances to be batched, while remaining consistent with the Kalman methods.

We begin with a brief discussion of the types of feedforward and recurrent network architectures that we are going to consider for training by EKF methods. We then discuss the *global* EKF training method, followed by recommendations for setting of parameters for EKF methods, including the relationship of the choice of learning rate to the initialization of the error covariance matrix. We then provide treatments of the decoupled extended Kalman filter (DEKF) method as well as the multistream procedure that can be applied with any level of decoupling. We discuss at length a variety of issues related to computer implementation, including derivative calculations, computationally efficient formulations, methods for avoiding matrix inversions, and square-root filtering for computational stability. This is followed by a number of special topics, including training with constrained weights and alternative cost functions. We then provide an overview of applications of EKF methods to a series of problems in control, diagnosis, and modeling of automotive powertrain systems. We conclude the chapter with a discussion of the virtues and limitations of EKF training methods, and provide a series of guidelines for implementation and use.

2.2 NETWORK ARCHITECTURES

We consider in this chapter two types of network architecture: the well-known feedforward layered network and its dynamic extension, the recurrent multilayered perceptron (RMLP). A block-diagram representa-

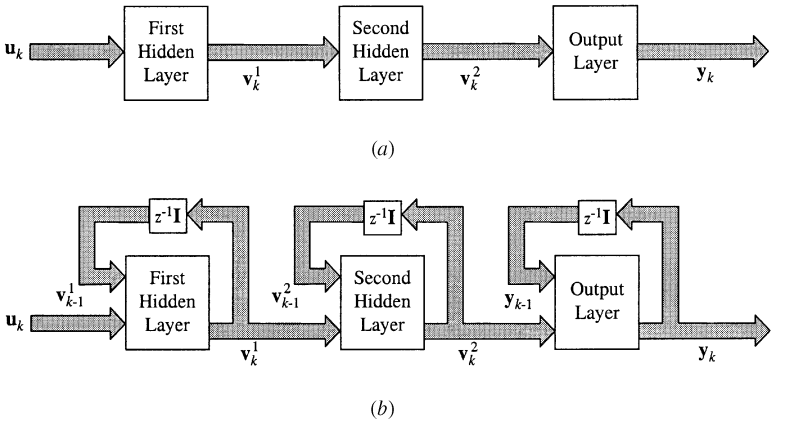

Figure 2.1 Block-diagram representation of two hidden layer networks. (*a*) depicts a feedforward layered neural network that provides a static mapping between the input vector $\mathbf{u}_k$ and the output vector $\mathbf{y}_k$. (*b*) depicts a recurrent multilayered perceptron (RMLP) with two hidden layers. In this case, we assume that there are time-delayed recurrent connections between the outputs and inputs of all nodes within a layer. The signals $\mathbf{v}_k^i$ denote the node activations for the *i*th layer. Both of these block representations assume that bias connections are included in the feedforward connections.

tion of these types of networks is given in Figure 2.1. Figure 2.2 shows an example network, denoted as a 3-3-3-2 network, with three inputs, two hidden layers of three nodes each, and an output layer of two nodes. Figure 2.3 shows a similar network, but modified to include interlayer, time-delayed recurrent connections. We denote this as a 3-3R-3R-2R RMLP, where the letter "R" denotes a recurrent layer. In this case, both hidden layers as well as the output layer are recurrent. The essential difference between the two types of networks is the recurrent network's ability to encode temporal information. Once trained, the feedforward

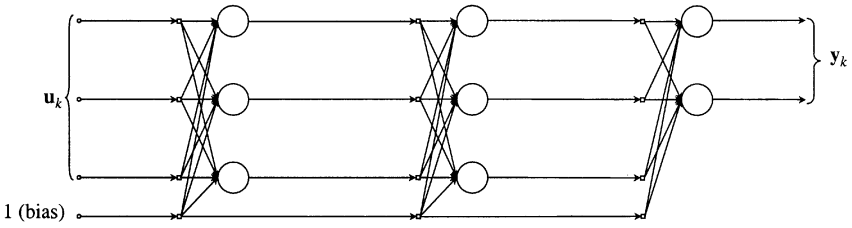

Figure 2.2 A schematic diagram of a 3-3-3-2 feedforward network architecture corresponding to the block diagram of Figure 2.1*a*.

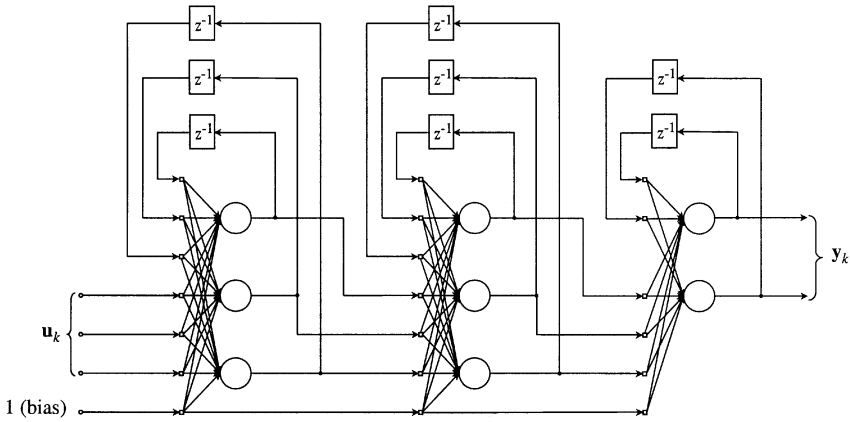

Figure 2.3. A schematic diagram of a 3-3R-3R-2R recurrent network architecture corresponding to the block diagram of Figure 2.1*b*. Note the presence of time delay operators and recurrent connections between the nodes of a layer.

network merely carries out a static mapping from input signals $\mathbf{u}_k$ to outputs $\mathbf{y}_k$, such that the output is independent of the history in which input signals are presented. On the other hand, a trained RMLP provides a dynamic mapping, such that the output $\mathbf{y}_k$ is not only a function of the current input pattern $\mathbf{u}_k$, but also implicitly a function of the entire history of inputs through the time-delayed recurrent node activations, given by the vectors $\mathbf{v}_{k-1}^i$, where i indexes layer number.

2.3 THE EKF PROCEDURE

We begin with the equations that serve as the basis for the derivation of the EKF family of neural network training algorithms. A neural network's behavior can be described by the following nonlinear discrete-time system:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \boldsymbol{\omega}_k \tag{2.1}$$

$$\mathbf{y}_k = \mathbf{h}_k(\mathbf{w}_k, \mathbf{u}_k, \mathbf{v}_{k-1}) + \boldsymbol{\nu}_k. \tag{2.2}$$

The first of these, known as the process equation, merely specifies that the state of the ideal neural network is characterized as a stationary process corrupted by process noise $\boldsymbol{\omega}_k$, where the state of the system is given by the network's weight parameter values $\mathbf{w}_k$. The second equation, known as the observation or measurement equation, represents the network's desired

response vector $\mathbf{y}_k$ as a nonlinear function of the input vector $\mathbf{u}_k$, the weight parameter vector $\mathbf{w}_k$, and, for recurrent networks, the recurrent node activations $\mathbf{v}_k$; this equation is augmented by random measurement noise $\boldsymbol{\nu}_k$. The measurement noise $\boldsymbol{\nu}_k$ is typically characterized as zero-mean, white noise with covariance given by $E[\boldsymbol{\nu}_k \boldsymbol{\nu}_l^T] = \delta_{k,l}\mathbf{R}_k$. Similarly, the process noise $\boldsymbol{\omega}_k$ is also characterized as zero-mean, white noise with covariance given by $E[\boldsymbol{\omega}_k \boldsymbol{\omega}_l^T] = \delta_{k,l}\mathbf{Q}_k$.

2.3.1 Global EKF Training

The training problem using Kalman filter theory can now be described as finding the minimum mean-squared error estimate of the state $\mathbf{w}$ using all observed data so far. We assume a network architecture with M weights and N_o output nodes and cost function components. The EKF solution to the training problem is given by the following recursion (see Chapter 1):

$$\mathbf{A}_k = [\mathbf{R}_k + \mathbf{H}_k^T \mathbf{P}_k \mathbf{H}_k]^{-1}, \tag{2.3}$$

$$\mathbf{K}_k = \mathbf{P}_k \mathbf{H}_k \mathbf{A}_k, \tag{2.4}$$

$$\hat{\mathbf{w}}_{k+1} = \hat{\mathbf{w}}_k + \mathbf{K}_k \boldsymbol{\xi}_k, \tag{2.5}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{K}_k \mathbf{H}_k^T \mathbf{P}_k + \mathbf{Q}_k. \tag{2.6}$$

The vector $\hat{\mathbf{w}}_k$ represents the estimate of the state (i.e., weights) of the system at update step k. This estimate is a function of the Kalman gain matrix $\mathbf{K}_k$ and the error vector $\boldsymbol{\xi}_k = \mathbf{y}_k - \hat{\mathbf{y}}_k$, where $\mathbf{y}_k$ is the target vector and $\hat{\mathbf{y}}_k$ is the network's output vector for the kth presentation of a training pattern. The Kalman gain matrix is a function of the approximate error covariance matrix $\mathbf{P}_k$, a matrix of derivatives of the network's outputs with respect to all trainable weight parameters $\mathbf{H}_k$, and a global scaling matrix $\mathbf{A}_k$. The matrix $\mathbf{H}_k$ may be computed via static backpropagation or backpropagation through time for feedforward and recurrent networks, respectively (described below in Section 2.6.1). The scaling matrix $\mathbf{A}_k$ is a function of the measurement noise covariance matrix $\mathbf{R}_k$, as well as of the matrices $\mathbf{H}_k$ and $\mathbf{P}_k$. Finally, the approximate error covariance matrix $\mathbf{P}_k$ evolves recursively with the weight vector estimate; this matrix encodes second derivative information about the training problem, and is augmented by the covariance matrix of the process noise $\mathbf{Q}_k$. This algorithm attempts to find weight values that minimize the sum of squared error $\sum_k \boldsymbol{\xi}_k^T \boldsymbol{\xi}_k$. Note that the algorithm requires that the measurement and

process noise covariance matrices, $\mathbf{R}_k$ and $\mathbf{Q}_k$, be specified for all training instances. Similarly, the approximate error covariance matrix $\mathbf{P}_k$ must be initialized at the beginning of training. We consider these issues below in Section 2.3.3.

GEKF training is carried out in a sequential fashion as shown in the signal flow diagram of Figure 2.4. One step of training involves the following steps:

1. An input training pattern $\mathbf{u}_k$ is propagated through the network to produce an output vector $\hat{\mathbf{y}}_k$. Note that the forward propagation is a function of the recurrent node activations $\mathbf{v}_{k-1}$ from the previous time step for RMLPs. The error vector $\boldsymbol{\xi}_k$ is computed in this step as well.
2. The derivative matrix $\mathbf{H}_k$ is obtained by backpropagation. In this case, there is a separate backpropagation for each component of the output vector $\hat{\mathbf{y}}_k$, and the backpropagation phase will involve a time history of recurrent node activations for RMLPs.
3. The Kalman gain matrix is computed as a function of the derivative matrix $\mathbf{H}_k$, the approximate error covariance matrix $\mathbf{P}_k$, and the measurement covariance noise matrix $\mathbf{R}_k$. Note that this step includes the computation of the global scaling matrix $\mathbf{A}_k$.
4. The network weight vector is updated using the Kalman gain matrix $\mathbf{K}_k$, the error vector $\boldsymbol{\xi}_k$, and the current values of the weight vector $\hat{\mathbf{w}}_k$.

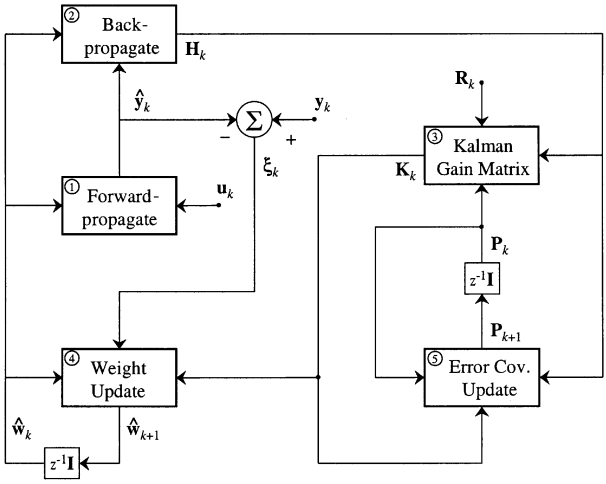

Figure 2.4 Signal flow diagram for EKF neural network training. The first two steps, comprising the forward- and backpropagation operations, will depend on whether or not the network being trained has recurrent connections. On the other hand, the EKF calculations encoded by steps (3)–(5) are independent of network type.

5. The approximate error covariance matrix is updated using the Kalman gain matrix $\mathbf{K}_k$, the derivative matrix $\mathbf{H}_k$, and the current values of the approximate error covariance matrix $\mathbf{P}_k$. Although not shown, this step also includes augmentation of the error covariance matrix by the covariance matrix of the process noise $\mathbf{Q}_k$.

2.3.2 Learning Rate and Scaled Cost Function

We noted above that $\mathbf{R}_k$ is the covariance matrix of the measurement noise and that this matrix must be specified for each training pattern. Generally speaking, training problems that are characterized by noisy measurement data usually require that the elements of $\mathbf{R}_k$ be scaled larger than for those problems with relatively noise-free training data. In [5, 7, 12], we interpret this measurement error covariance matrix to represent an inverse learning rate: $\mathbf{R}_k = \eta_k^{-1}\mathbf{S}_k^{-1}$, where the training cost function at time step k is now given by $\mathscr{E}_k = \frac{1}{2}\boldsymbol{\xi}_k^T\mathbf{S}_k\boldsymbol{\xi}_k$, and $\mathbf{S}_k$ allows the various network output components to be scaled nonuniformly. Thus, the global scaling matrix $\mathbf{A}_k$ of equation (2.3) can be written as

$$\mathbf{A}_k = \left[\frac{1}{\eta_k}\mathbf{S}_k^{-1} + \mathbf{H}_k^T\mathbf{P}_k\mathbf{H}_k\right]^{-1}. \tag{2.7}$$

The use of the weighting matrix $\mathbf{S}_k$ in Eq. (2.7) poses numerical difficulties when the matrix is singular.[1] We reformulate the GEKF algorithm to eliminate this difficulty by distributing the square root of the weighting matrix into both the derivative matrices as $\mathbf{H}_k^* = \mathbf{H}_k\mathbf{S}_k^{1/2}$ and the error vector as $\boldsymbol{\xi}_k^* = \mathbf{S}_k^{1/2}\boldsymbol{\xi}_k$. The matrices $\mathbf{H}_k^*$ thus contain the scaled derivatives of network outputs with respect to the weights of the network. The rescaled extended Kalman recursion is then given by

$$\mathbf{A}_k^* = \left[\frac{1}{\eta_k}\mathbf{I} + (\mathbf{H}_k^*)^T\mathbf{P}_k\mathbf{H}_k^*\right]^{-1}, \tag{2.8}$$

$$\mathbf{K}_k^* = \mathbf{P}_k\mathbf{H}_k^*\mathbf{A}_k^*, \tag{2.9}$$

$$\hat{\mathbf{w}}_{k+1} = \hat{\mathbf{w}}_k + \mathbf{K}_k^*\boldsymbol{\xi}_k^*, \tag{2.10}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{K}_k^*(\mathbf{H}_k^*)^T\mathbf{P}_k + \mathbf{Q}_k. \tag{2.11}$$

Note that this rescaling does not change the evolution of either the weight vector or the approximate error covariance matrix, and eliminates the need

[1]This may occur when we utilize penalty functions to impose explicit constraints on network outputs. For example, when a constraint is not violated, we set the corresponding diagonal element of $\mathbf{S}_k$ to zero, thereby rendering the matrix singular.

to compute the inverse of the weighting matrix $\mathbf{S}_k$ for each training pattern. For the sake of clarity in the remainder of this chapter, we shall assume a uniform scaling of output signals, $\mathbf{S}_k = \mathbf{I}$, which implies $\mathbf{R}_k = \eta_k^{-1}\mathbf{I}$, and drop the asterisk notation.

2.3.3 Parameter Settings

EKF training algorithms require the setting of a number of parameters. In practice, we have employed the following rough guidelines. First, we typically assume that the input–output data have been scaled and transformed to reasonable ranges (e.g., zero mean, unit variance for all continuous input and output variables). We also assume that weight values are initialized to small random values drawn from a zero-mean uniform or normal distribution. The approximate error covariance matrix is initialized to reflect the fact that no a priori knowledge was used to initialize the weights; this is accomplished by setting $\mathbf{P}_0 = \epsilon^{-1}\mathbf{I}$, where ϵ is a small number (of the order of 0.001–0.01). As noted above, we assume uniform scaling of outputs: $\mathbf{S}_k = \mathbf{I}$. Then, training data that are characterized by noisy measurements usually require small values for the learning rate η_k to achieve good training performance; we typically bound the learning rate to values between 0.001 and 1. Finally, the covariance matrix $\mathbf{Q}_k$ of the process noise is represented by a scaled identity matrix $q_k\mathbf{I}$, with the scale factor q_k ranging from as small as zero (to represent no process noise) to values of the order of 0.1. This factor is generally annealed from a large value to a limiting value of the order of 10^{-6}. This annealing process helps to accelerate convergence and, by keeping a nonzero value for the process noise term, helps to avoid divergence of the error covariance update in Eqs. (2.6) and (2.11).

We show here that the setting of the learning rate, the process noise covariance matrix, and the initialization of the approximate error covariance matrix are interdependent, and that an arbitrary scaling can be applied to $\mathbf{R}_k$, $\mathbf{P}_k$, and $\mathbf{Q}_k$ without altering the evolution of the weight vector $\hat{\mathbf{w}}$ in Eqs. (2.5) and (2.10). First consider the Kalman gain of Eqs. (2.4) and (2.9). An arbitrary positive scaling factor μ can be applied to $\mathbf{R}_k$ and $\mathbf{P}_k$ without altering the contents of $\mathbf{K}_k$:

$$\begin{aligned}
\mathbf{K}_k &= \mathbf{P}_k\mathbf{H}_k[\mathbf{R}_k + \mathbf{H}_k^T\mathbf{P}_k\mathbf{H}_k]^{-1} \\
&= \mu\mathbf{P}_k\mathbf{H}_k[\mu\mathbf{R}_k + \mathbf{H}_k^T\mu\mathbf{P}_k\mathbf{H}_k]^{-1} \\
&= \mathbf{P}_k^{\dagger}\mathbf{H}_k[\mathbf{R}_k^{\dagger} + \mathbf{H}_k^T\mathbf{P}_k^{\dagger}\mathbf{H}_k]^{-1} \\
&= \mathbf{P}_k^{\dagger}\mathbf{H}_k\mathbf{A}_k^{\dagger},
\end{aligned}$$

where we have defined $\mathbf{R}_k^\dagger = \mu\mathbf{R}_k$, $\mathbf{P}_k^\dagger = \mu\mathbf{P}_k$, and $\mathbf{A}_k^\dagger = \mu^{-1}\mathbf{A}_k$. Similarly, the approximate error covariance update becomes

$$\begin{aligned}\mathbf{P}_{k+1}^\dagger &= \mu\mathbf{P}_{k+1} \\ &= \mu\mathbf{P}_k - \mathbf{K}_k\mathbf{H}_k^T\mu\mathbf{P}_k + \mu\mathbf{Q}_k \\ &= \mathbf{P}_k^\dagger - \mathbf{K}_k\mathbf{H}_k^T\mathbf{P}_k^\dagger + \mathbf{Q}_k^\dagger.\end{aligned}$$

This implies that a training trial characterized by the parameter settings $\mathbf{R}_k = \eta^{-1}\mathbf{I}$, $\mathbf{P}_0 = \epsilon^{-1}\mathbf{I}$, and $\mathbf{Q}_k = q\mathbf{I}$, would behave identically to a training trial with scaled versions of these parameter settings: $\mathbf{R}_k = \mu\eta^{-1}\mathbf{I}$, $\mathbf{P}_0 = \mu\epsilon^{-1}\mathbf{I}$, and $\mathbf{Q}_k = \mu q\mathbf{I}$. Thus, for any given EKF training problem, there is no one best set of parameter settings, but a continuum of related settings that must take into account the properties of the training data for good performance. This also implies that only two effective parameters need to be set. Regardless of the training problem considered, we have typically chosen the initial error covariance matrix to be $\mathbf{P}_0 = \epsilon^{-1}\mathbf{I}$, with $\epsilon = 0.01$ and 0.001 for sigmoidal and linear activation functions, respectively. This leaves us to specify values for η_k and $\mathbf{Q}_k$, which are likely to be problem-dependent.

2.4 DECOUPLED EKF (DEKF)

The computational requirements of GEKF are dominated by the need to store and update the approximate error covariance matrix $\mathbf{P}_k$ at each time step. For a network architecture with N_o outputs and M weights, GEKF's computational complexity is $\mathcal{O}(N_o M^2)$ and its storage requirements are $\mathcal{O}(M^2)$. The parameter-based DEKF algorithm is derived from GEKF by assuming that the interactions between certain weight estimates can be ignored. This simplification introduces many zeroes into the matrix $\mathbf{P}_k$. If the weights are decoupled so that the weight groups are mutually exclusive of one another, then $\mathbf{P}_k$ can be arranged into block-diagonal form. Let g refer to the number of such weight groups. Then, for group i, the vector $\hat{\mathbf{w}}_k^i$ refers to the estimated weight parameters, $\mathbf{H}_k^i$ is the submatrix of derivatives of network outputs with respect to the ith group's weights, $\mathbf{P}_k^i$ is the weight group's approximate error covariance matrix, and $\mathbf{K}_k^i$ is its Kalman gain matrix. The concatenation of the vectors $\hat{\mathbf{w}}_k^i$ forms the vector $\hat{\mathbf{w}}_k$. Similarly, the global derivative matrix $\mathbf{H}_k$ is composed via concatena-

tion of the individual submatrices $\mathbf{H}_k^i$. The DEKF algorithm for the ith weight group is given by

$$\mathbf{A}_k = \left[\mathbf{R}_k + \sum_{j=1}^{g} (\mathbf{H}_k^j)^T \mathbf{P}_k^j \mathbf{H}_k^j \right]^{-1}, \tag{2.12}$$

$$\mathbf{K}_k^i = \mathbf{P}_k^i \mathbf{H}_k^i \mathbf{A}_k, \tag{2.13}$$

$$\hat{\mathbf{w}}_{k+1}^i = \hat{\mathbf{w}}_k^i + \mathbf{K}_k^i \boldsymbol{\xi}_k, \tag{2.14}$$

$$\mathbf{P}_{k+1}^i = \mathbf{P}_k^i - \mathbf{K}_k^i (\mathbf{H}_k^i)^T \mathbf{P}_k^i + \mathbf{Q}_k^i. \tag{2.15}$$

A single global sealing matrix $\mathbf{A}_k$, computed with contributions from all of the approximate error covariance matrices and derivative matrices, is used to compute the Kalman gain matrices, $\mathbf{K}_k^i$. These gain matrices are used to update the error covariance matrices for all weight groups, and are combined with the global error vector $\boldsymbol{\xi}_k$ for updating the weight vectors. In the limit of a single weight group ($g = 1$), the DEKF algorithm reduces exactly to the GEKF algorithm.

The computational complexity and storage requirements for DEKF can be significantly less than those of GEKF. For g disjoint weight groups, the computational complexity of DEKF becomes $\mathcal{O}(N_o^2 M + N_o \sum_{i=1}^{g} M_i^2)$, where M_i is the number of weights in group i, while the storage requirements become $\mathcal{O}(\sum_{i=1}^{g} M_i^2)$. Note that this complexity analysis does not include the computational requirements for the matrix of derivatives, which is independent of the level of decoupling. It should be noted that in the case of training recurrent networks or networks as feedback controllers, the computational complexity of the derivative calculations can be significant.

We have found that decoupling of the weights of the network by node (i.e., each weight group is composed of a single node's weight) is rather natural and leads to compact and efficient computer implementations. Furthermore, this level of decoupling typically exhibits substantial computational savings relative to GEKF, often with little sacrifice in network performance after completion of training. We refer to this level of decoupling as *node-decoupled* EKF or NDEKF. Other forms of decoupling considered have been *fully decoupled* EKF, in which each individual weight constitutes a unique group (thereby resulting in an error covariance matrix that has diagonal structure), and *layer-decoupled* EKF, in which weights are grouped by the layer to which they belong [13]. We show an example of the effect of all four levels of decoupling on the structure of

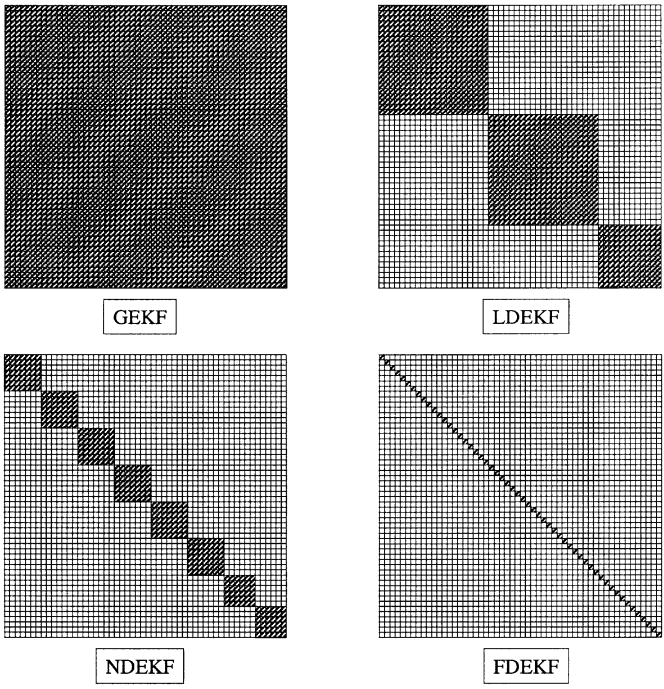

Figure 2.5 Block-diagonal representation of the approximate error covariance matrix $\mathbf{P}_k$ for the RMLP network shown in Figure 2.3 for four different levels of decoupling. This network has two recurrent layers with three nodes each and each node with seven incoming connections. The output layer is also recurrent, but its two nodes only have six connections each. Only the shaded portions of these matrices are updated and maintained for the various forms of decoupling shown. Note that we achieve a reduction by nearly a factor of 8 in computational complexity for the case of node decoupling relative to GEKF in this example.

the approximate error covariance matrix in Figure 2.5. For the remainder of this chapter, we explicitly consider only two different levels of decoupling for EKF training: global and node-decoupled EKF.

2.5 MULTISTREAM TRAINING

Up to this point, we have considered forms of EKF training in which a single weight-vector update is performed on the basis of the presentation of a single input–output training pattern. However, there may be situations for which a *coordinated* weight update, on the basis of multiple training

patterns, would be advantageous. We consider in this section an abstract example of such a situation, and describe the means by which the EKF method can be naturally extended to simultaneously handle multiple training instances for a single weight update.[2]

Consider the standard recurrent network training problem: 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 external 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. Alternatively, a sequence of outputs that centers about one level may be followed by one that centers about a different level. In any case, the tendency always exists in a straightforward training process 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 if the data are repeatedly presented in the same order.

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 would involve 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 [10–12] is based on the principle that each weight update should attempt to satisfy simultaneously the demands from multiple input–output pairs. However, it retains the useful stochastic aspects of sequential updating, and requires much less computation time between updates. We now describe the mechanics of multistream training.

[2]In the case of purely linear systems, there is no advantage in batching up a collection of training instances for a single weight update via Kalman filter methods, since all weight updates are completely consistent with previously observed data. On the other hand, derivative calculations and the extended Kalman recursion for nonlinear networks utilize first-order approximations, so that weight updates are no longer guaranteed to be consistent with all previously processed data.

In a typical training problem, we deal with one or more files, each of which contains a sequence of data. Breaking the overall data 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*. In the multistream procedure we progress sequentially 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 backpropagation through time (BPTT(h)) as discussed in Section 2.6.1 below. 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, 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 gradient updates were being used, we would simply average the updates that would have 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 $N_o N_s$ 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, because of the global scaling matrix $\mathbf{A}_k$, is then to produce weight updates that 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. Note that we are still minimizing the same sum of squared error cost function.

In single-stream EKF training, we place derivatives of network outputs with respect to network weights in the matrix $\mathbf{H}_k$ constructed from N_o 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 $N_o N_s$. Similarly, the vector of errors $\boldsymbol{\xi}_k$ has $N_o N_s$ elements. Apart from these augmentations of $\mathbf{H}_k$ and $\boldsymbol{\xi}_k$, the form of the Kalman recursion is unchanged.

Given these considerations, we define the decoupled multistream EKF recursion as follows. We shall alter the temporal indexing by specifying a range of training patterns that indicate how the multi-stream recursion should be interpreted. We define $l = k + N_s - 1$ and allow the range $k:l$ to specify the batch of training patterns for which a single weight vector update will be performed. Then, the matrix $\mathbf{H}^i_{k:l}$ is the concatenation of the derivative matrices for the ith group of weights and for training patterns that have been assigned to the range $k:l$. Similarly, the augmented error vector is denoted by $\boldsymbol{\xi}_{k:l}$. We construct the derivative matrices and error vector, respectively, by

$$\mathbf{H}_{k:l} = (\mathbf{H}_k \mathbf{H}_{k+1} \mathbf{H}_{k+2} \cdots \mathbf{H}_{l-1} \mathbf{H}_l),$$
$$\boldsymbol{\xi}_{k:l} = (\boldsymbol{\xi}_k^T \boldsymbol{\xi}_{k+1}^T \boldsymbol{\xi}_{k+2}^T \cdots \boldsymbol{\xi}_{l-1}^T \boldsymbol{\xi}_l^T)^T.$$

We use a similar notation for the measurement error covariance matrix $\mathbf{R}_{k:l}$ and the global scaling matrix $\mathbf{A}_{k:l}$, both square matrices of dimension $N_o N_s$, and for the Kalman gain matrices $\mathbf{K}^i_{k:l}$, with size $M_i \times N_o N_s$. The multistream DEKF recursion is then given by

$$\mathbf{A}_{k:l} = \left[\mathbf{R}_{k:l} + \sum_{j=1}^{g} (\mathbf{H}^j_{k:l})^T \mathbf{P}^j_k \mathbf{H}^j_{k:l} \right]^{-1}, \tag{2.16}$$

$$\mathbf{K}^i_{k:l} = \mathbf{P}^i_k \mathbf{H}^i_{k:l} \mathbf{A}_{k:l}, \tag{2.17}$$

$$\hat{\mathbf{w}}^i_{k+N_s} = \hat{\mathbf{w}}^i_k + \mathbf{K}^i_{k:l} \boldsymbol{\xi}_{k:l}, \tag{2.18}$$

$$\mathbf{P}^i_{k+N_s} = \mathbf{P}^i_k - \mathbf{K}^i_{k:l} (\mathbf{H}^i_{k:l})^T \mathbf{P}^i_k + \mathbf{Q}^i_k. \tag{2.19}$$

Note that this formulation reduces correctly to the original DEKF recursion in the limit of a single stream, and that multistream GEKF is given in the case of a single weight group. We provide a block diagram representation of the multistream GEKF procedure in Figure 2.6. Note that the steps of training are very similar to the single-stream case, with the exception of multiple forward-propagation and backpropagation steps, and the concatenation operations for the derivative matrices and error vectors.

Let us consider the computational implications of the multistream method. The sizes of the approximate error covariance matrices $\mathbf{P}_k^i$ and the weight vectors $\mathbf{w}_k^i$ are independent of the chosen number of streams. On the other hand, we noted above the increase in size for the derivative matrices $\mathbf{H}_{k:l}^i$, as well as of the Kalman gain matrices $\mathbf{K}_{k:l}^i$. However, the computation required to obtain $\mathbf{H}_{k:l}^i$ and to compute updates to $\mathbf{P}_k^i$ is the same as for N_s separate updates. The major additional computational burden is the inversion required to obtain the matrix $\mathbf{A}_{k:l}$ whose dimension is N_s times larger than in the single-stream case. Even this cost tends to be small compared with that associated with the $\mathbf{P}_k^i$ matrices, as long as

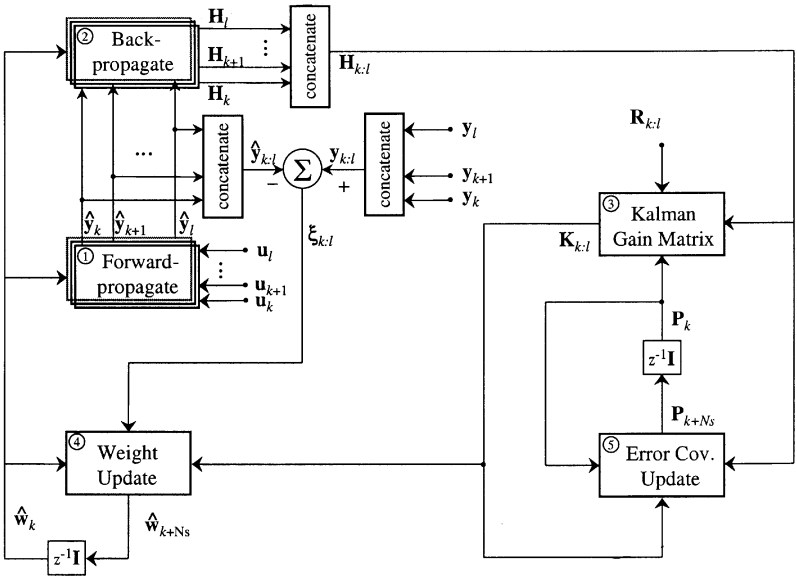

Figure 2.6 Signal flow diagram for multistream EKF neural network training. The first two steps are comprised of multiple forward- and backpropagation operations, determined by the number of streams N_s selected; these steps also depend on whether or not the network being trained has recurrent connections. On the other hand, once the derivative matrix $\mathbf{H}_{k:l}$ and error vector $\boldsymbol{\xi}_{k:l}$ are formed, the EKF steps encoded by steps (3)–(5) are independent of number of streams and network type.

$N_o N_s$ 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 $\mathbf{A}_{k:l}$ impractical, the inversion may be avoided by using one of the alternative EKF formulations described below in Section 2.6.3.

2.5.1 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 the recursive least-squares (RLS) algorithm. Assume that a training data set is represented by m unique training patterns. The kth training pattern is represented by a d-dimensional input vector $\mathbf{u}_k$, where we assume that all input vectors include a constant bias component of value equal to 1, and a 1-dimensional output target y_k. The simple linear model for this system is given by

$$\hat{y}_k = \mathbf{u}_k^T \mathbf{w}_f, \tag{2.20}$$

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

$$a_k = [1 + \mathbf{u}_k^T \mathbf{P}_k \mathbf{u}_k]^{-1}, \tag{2.21}$$

$$\mathbf{k}_k = \mathbf{P}_k \mathbf{u}_k a_k, \tag{2.22}$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{k}_k (y_k - \hat{y}_k), \tag{2.23}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{k}_k \mathbf{u}_k^T \mathbf{P}_k, \tag{2.24}$$

where the diagonal elements of $\mathbf{P}_0$ are initialized to large positive values, and $\mathbf{w}_0$ to a vector of small random values. Also, $\mathbf{w}_f = \mathbf{w}_m$ after a single presentation of all training data (i.e., after a single epoch).

We recover a batch, least-squares 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 $\mathbf{U}$ 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 column vector $\mathbf{y}$,

where elements of $\mathbf{y}$ are ordered identically with the matrix $\mathbf{U}$. As before, we select the initial weight vector $\mathbf{w}_0$ to consist of randomly chosen values, and we select $\mathbf{P}_0 = \epsilon^{-1}\mathbf{I}$, with ϵ small. 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

$$\hat{\mathbf{y}}_0 = \mathbf{U}^T\mathbf{w}_0. \tag{2.25}$$

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

$$\mathbf{A}_0 = [\mathbf{I} + \mathbf{U}^T\mathbf{P}_0\mathbf{U}]^{-1}, \tag{2.26}$$

$$\mathbf{K}_0 = \mathbf{P}_0\mathbf{U}\mathbf{A}_0, \tag{2.27}$$

$$\mathbf{w}_1 = \mathbf{w}_0 + \mathbf{K}_0(\mathbf{y} - \hat{\mathbf{y}}_0), \tag{2.28}$$

where we have chosen not to include the error covariance update here for reasons that will soon become clear. At the beginning of training, we recognize that $\mathbf{P}_0$ is large, and we assume that the training data set is scaled so that $\mathbf{U}^T\mathbf{P}_0\mathbf{U} \gg \mathbf{I}$. This allows $\mathbf{A}_0$ to be approximated by

$$\mathbf{A}_0 \approx \epsilon[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}, \tag{2.29}$$

since $\mathbf{P}_0$ is diagonal. Given this approximation, we can write the Kalman gain matrix as

$$\mathbf{K}_0 = \mathbf{U}[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}. \tag{2.30}$$

We now substitute Eqs. (2.25) and (2.30) into Eq. (2.28) to derive the weight vector after one time step of this m-stream Kalman filter procedure:

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{w}_0 + \mathbf{U}[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}[\mathbf{y} - \mathbf{U}^T\mathbf{w}_0] \\ &= \mathbf{w}_0 - \mathbf{U}[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}\mathbf{U}^T\mathbf{w}_0 + \mathbf{U}[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}\mathbf{y}.\end{aligned} \tag{2.31}$$

If we apply the matrix equality $\lim_{\epsilon\to 0}\mathbf{U}[\epsilon\mathbf{I} + \mathbf{U}^T\mathbf{U}]^{-1}\mathbf{U}^T = \mathbf{I}$, we obtain the pseudoinverse solution:

$$\mathbf{w}_f = \mathbf{w}_1 = [\mathbf{U}\mathbf{U}^T]^{-1}\mathbf{U}\mathbf{y}, \tag{2.32}$$

where we have made use of

$$\lim_{\epsilon \to 0} \mathbf{U}[\epsilon \mathbf{I} + \mathbf{U}^T \mathbf{U}]^{-1} \mathbf{U}^T = \mathbf{I}, \tag{2.33}$$

$$\lim_{\epsilon \to 0} \mathbf{U}[\epsilon \mathbf{I} + \mathbf{U}^T \mathbf{U}]^{-1} \mathbf{U}^T = [\mathbf{U}\mathbf{U}^T]^{-1} \mathbf{U}\mathbf{U}^T, \tag{2.34}$$

$$\lim_{\epsilon \to 0} \mathbf{U}[\epsilon \mathbf{I} + \mathbf{U}^T \mathbf{U}]^{-1} = [\mathbf{U}\mathbf{U}^T]^{-1} \mathbf{U}. \tag{2.35}$$

Thus, one step of the multistream Kalman recursion recovers very closely the least-squares 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 is coordinated through the global scaling matrix $\mathbf{A}$. It is intuitively clear that this coordination is most valuable when the various streams place contrasting demands on the network.

2.5.2 Advantages and Extensions of Multistream Training

Discussions of the training of networks with external recurrence often distinguish between series–parallel and parallel configurations. 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 networks with internal recurrence (e.g., recurrent multilayered perceptrons), 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, that is, from the same training task. Indeed, we have demonstrated that a single fixed-weight, recurrent neural network, trained by multistream EKF, can carry out multiple tasks in a control context, namely, to act as a stabilizing controller for multiple distinct and unrelated systems, without explicit knowledge of system identity [14]. This work demonstrated that the trained network was capable of exhibiting what could be considered to be adaptive behavior: the network, acting as a controller, observed the behavior of the system (through the system's output), implicitly identified which system the network was being subjected to, and then took actions to stabilize the system. We view this somewhat unexpected behavior as being the direct result of combining an effective training procedure with enabling representational capabilities that recurrent networks provide.

2.6 COMPUTATIONAL CONSIDERATIONS

We discuss here a number of topics related to implementation of the various EKF training procedures from a computational perspective. In particular, we consider issues related to computation of derivatives that are critical to the EKF methods, followed by discussions of computationally efficient formulations, methods for avoiding matrix inversions, and the use of square-root filtering as an alternative means of insuring stable performance.

2.6.1 Derivative Calculations

We discussed above both the global and decoupled versions of the EKF algorithm, where we consider the global EKF to be a limiting form of decoupled EKF (i.e., DEKF with a single weight group). In addition, we have described the multistream EKF procedure as a means of batching training instances, and have noted that multistreaming can be used with any form of decoupled EKF training, for both feedforward and recurrent networks. The various EKF procedures can all be compactly described by the DEKF recursion of Eqs. (2.12)–(2.15), where we have assumed that the derivative matrices $\mathbf{H}_k^i$ are given. However, the implications for computationally efficient and clear implementations of the various forms of EKF training depend upon the derivative calculations, which are dictated by whether a network architecture is static or dynamic (i.e., feedforward or recurrent), and whether or not multistreaming is used. Here

we provide insight into the nature of derivative calculations for training of both static and dynamic networks with EKF methods (see [12] for implementation details).

We assume the convention that a network's weights are organized by node, regardless of the degree of decoupling, which allows us to naturally partition the matrix of derivatives of network outputs with respect to weight parameters, $\mathbf{H}_k$, into a set of G submatrices $\mathbf{H}_k^i$, where G is the number of nodes of the network. Then, each matrix $\mathbf{H}_k^i$ denotes the matrix of derivatives of network outputs with respect to the weights associated with the ith node of the network. For feedforward networks, these submatrices can be written as the outer product of two vectors [3],

$$\mathbf{H}_k^i = \mathbf{u}_k^i (\boldsymbol{\psi}_k^i)^T,$$

where $\mathbf{u}_k^i$ is the ith node's input vector and $\boldsymbol{\psi}_k^i$ is a vector of partial derivatives of the network's outputs with respect to the ith node's net input, defined as the dot product of the weight vector $\mathbf{w}_k^i$ with the corresponding input vector $\mathbf{u}_k^i$. Note that the vectors $\boldsymbol{\psi}_k^i$ are computed via the backpropagation process, where the dimension of each of these vectors is determined by the number of network outputs. In contrast to the standard backpropagation algorithm, which begins the derivative calculation process (i.e., backpropagation) with error signals for each of the network's outputs, and effectively combines these error signals (for multiple-output problems) during the backpropagation process, the EKF methods begin the process with signals of unity for each network output and backpropagate a separate signal for each unique network output.

In the case of recurrent networks, we assume the use of truncated backpropagation through time for calculation of derivatives, with a truncation depth of h steps; this process is denoted by BPTT(h). Now, each submatrix $\mathbf{H}_k^i$ can no longer be expressed as a simple outer product of two vectors; rather, each of these submatrices is expressed as the sum of a series of outer products:

$$\mathbf{H}_k^i = \sum_{j=1}^{h} \mathbf{H}_k^{i,j} = \sum_{j=1}^{h} \mathbf{u}_k^{i,j} (\boldsymbol{\psi}_k^{i,j})^T,$$

where the matrix $\mathbf{H}_k^{i,j}$ is the contribution from the jth step of backpropagation to the computation of the total derivative matrix for the ith node; the vector $\mathbf{u}_k^{i,j}$ is the vector of inputs to the ith node at the jth step of backpropagation; and $\boldsymbol{\psi}_k^{i,j}$ is the vector of backpropagated derivatives of

network outputs with respect to the ith node's net input at the jth step of backpropagation. Here, we have chosen arbitrarily to have j increase as we step back in time.

Finally, consider multi-stream training, where we assume that the training problem involves a recurrent network architecture, with derivatives computed by BPTT(h) (feedforward networks are subsumed by the case of $h = 1$). We again assume an N_o-component cost function with N_s streams, and define $l = k + N_s - 1$. Then, each submatrix $\mathbf{H}^i_{k:l}$ becomes a concatenation of a series of submatrices, each of which is expressed as the sum of a series of outer products:

$$\mathbf{H}^i_{k:l} = \left[\sum_{j=1}^{h} \mathbf{H}_k^{i,j,1} \sum_{j=1}^{h} \mathbf{H}_k^{i,j,2} \cdots \sum_{j=1}^{h} \mathbf{H}_k^{i,j,N_s} \right] \tag{2.36}$$

$$= \left[\sum_{j=1}^{h} \mathbf{u}_k^{i,j,1} (\boldsymbol{\psi}_k^{i,j,1})^T \sum_{j=1}^{h} \mathbf{u}_k^{i,j,2} (\boldsymbol{\psi}_k^{i,j,2})^T \cdots \sum_{j=1}^{h} \mathbf{u}_k^{i,j,N_s} (\boldsymbol{\psi}_k^{i,j,N_s})^T \right]. \tag{2.37}$$

Here we have expressed each submatrix $\mathbf{H}^i_{k:l}$ as an $M_i \times (N_o N_s)$ matrix, where M_i is the number of weights corresponding to the networks ith node. The submatrices $\mathbf{H}_k^{i,j,m}$ are of size $M_i \times N_o$, corresponding to a single training stream. For purposes of a compact representation, we express each matrix $\mathbf{H}^i_{k:l}$ as a sum of matrices (as opposed to a concatenation) by forming vectors $\boldsymbol{\Psi}_k^{i,j,m}$ from the vectors $\boldsymbol{\psi}_k^{i,j,m}$ in the following fashion. The vector $\boldsymbol{\Psi}_k^{i,j,m}$ is of length $N_o N_s$ with components set to zero everywhere except for in the mth (out of N_s) block of length N_o, where this subvector is set equal to the vector $\boldsymbol{\psi}_k^{i,j,m}$. Then,

$$\mathbf{H}^i_{k:l} = \sum_{j=1}^{h} \sum_{m=1}^{N_s} \mathbf{u}_k^{i,j,m} (\boldsymbol{\Psi}_k^{i,j,m})^T.$$

Note that the matrix is expressed in this fashion for notational convenience and consistency, and that we would make use of the sparse nature of the vector $\boldsymbol{\Psi}_k^{i,j,m}$ in implementation.

2.6.2 Computationally Efficient Formulations for Multiple-Output Problems

We now consider implications for the computational complexity of EKF training due to expressing the derivative calculations as a series of vector

outer products as shown above. We consider the simple case of feedforward networks trained by node-decoupled EKF (NDEKF) in which each node's weights comprise a unique group for purposes of the error covariance update. The NDEKF recursion can then be written as

$$\mathbf{A}_k = \left[\mathbf{R}_k + \sum_{j=1}^{G} \alpha_k^j \boldsymbol{\psi}_k^j (\boldsymbol{\psi}_k^j)^T \right]^{-1}, \tag{2.38}$$

$$\mathbf{K}_k^i = \mathbf{v}_k^i (\boldsymbol{\gamma}_k^i)^T, \tag{2.39}$$

$$\hat{\mathbf{w}}_{k+1}^i = \hat{\mathbf{w}}_k^i + [(\boldsymbol{\psi}_k^i)^T (\mathbf{A}_k \boldsymbol{\xi}_k)] \mathbf{v}_k^i, \tag{2.40}$$

$$\mathbf{P}_{k+1}^i = \mathbf{P}_k^i - \beta_k^i \mathbf{v}_k^i (\mathbf{v}_k^i)^T + \mathbf{Q}_k^i, \tag{2.41}$$

where we have used the following equations in intermediate steps:

$$\mathbf{v}_k^i = \mathbf{P}_k^i \mathbf{u}_k^i, \tag{2.42}$$

$$\boldsymbol{\gamma}_k^i = \mathbf{A}_k \boldsymbol{\psi}_k^i, \tag{2.43}$$

$$\alpha_k^i = (\mathbf{u}_k^i)^T \mathbf{v}_k^i, \tag{2.44}$$

$$\beta_k^i = (\boldsymbol{\gamma}_k^i)^T \boldsymbol{\psi}_k^i. \tag{2.45}$$

Based upon this partitioning of the derivative matrix $\mathbf{H}_k$, we find that the computational complexity of NDEKF is reduced from $\mathcal{O}(N_o^2 M + N_o \sum_{i=1}^{G} M_i^2)$ to $\mathcal{O}(N_o^2 G + \sum_{i=1}^{G} M_i^2)$, indicating a distinct advantage for feedforward networks with multiple output nodes. On the other hand, the partitioning of the derivative matrix does not provide any computational advantage for GEKF training of feedforward networks.

2.6.3 Avoiding Matrix Inversions

A complicating factor for effective implementation of EKF training schemes is the need to perform matrix inversions for those problems with multiple cost function components. We typically perform these types of calculations with matrix inversion routines based on singular-value decomposition [15]. Although these techniques have served us well over the years, we recognize that this often discourages "quick-and-dirty" implementations and may pose a large obstacle to hardware implementation.

Two classes of methods have been developed that allow EKF training to be performed for multiple-output problems without explicitly resorting to matrix inversion routines. The first class [16] depends on the partitioning

of the derivative matrices described above. This method computes the global scaling matrix $\mathbf{A}_k$ by recursively applying the matrix inversion lemma. This procedure provides results that are mathematically identical to conventional matrix inversion procedures, regardless of the degree of decoupling employed. In addition, it can be employed for training of any form of network, static or dynamic, as well as for the multistream procedure. On the other hand, we have found that this method often requires the use of double-precision arithmetic to produce results that are statistically identical to EKF implementations based on explicit matrix inversion methods.

The second class, developed by Plumer [17], treats each output component individually in an iterative procedure. This sequential update procedure accumulates the weight vector update as each output component is processed, and only applies the weight vector update after all output signals have been processed. The error covariance matrix is updated in a sequential fashion. Plumer's sequential-update form of EKF turns out to be exactly equivalent to the batch form of GFKF given above in which all output signals are processed simultaneously. However, for decoupled EKF training, it turns out that sequential updates only approximate the updates obtained via the simultaneous DEKF recursion of Eqs. (2.12)–(2.15), though this has been reported to not pose any problems during training.

The sequential DEKF method is compactly given by a set of equations that are similar to the simultaneous DEKF equations. We again assume a decoupling with g mutually exclusive groups of weights, with a limit of $g = 1$ reducing to the global version, and use the superscript i to refer to the individual weight groups. We handle the multistream case by labeling each cost function component from $l = 1$ to $N_o N_s$, where N_o and N_s refer to the number of network outputs and number of processing streams, respectively. A single weight vector update with the sequential multistream DEKF procedure requires an initialization step of $\Delta\hat{\mathbf{w}}^i_{k,0} = \mathbf{0}$ and $\mathbf{P}^i_{k,0} = \mathbf{P}^i_k$, where $\Delta\hat{\mathbf{w}}^i_{k,l}$ is used to accumulate the update to the weight vector. Then, the sequential multistream DEKF procedure is compactly represented by the following equations:

$$a_{k,l} = \left[r_{k,l} + \sum_{j=1}^{g} (\mathbf{h}^i_{k,l})^T \mathbf{P}^i_{k,l-1} \mathbf{h}^i_{k,l} \right]^{-1}, \tag{2.46}$$

$$\mathbf{k}^i_{k,l} = \mathbf{P}^i_{k,l-1} \mathbf{h}^i_{k,l} a_{k,l}, \tag{2.47}$$

$$\Delta\hat{\mathbf{w}}^i_{k,l} = \Delta\hat{\mathbf{w}}^i_{k,l-1} + \mathbf{k}^i_{k,l} \xi_{k,l} - \mathbf{k}^i_{k,l} [(\mathbf{h}^i_{k,l})^T \Delta\hat{\mathbf{w}}^i_{k,l-1}], \tag{2.48}$$

$$\mathbf{P}^i_{k,l} = \mathbf{P}^i_{k,l-1} - \mathbf{k}^i_{k,l} (\mathbf{h}^i_{k,l})^T \mathbf{P}^i_{k,l-1}. \tag{2.49}$$

Note that the scalar $r_{k,l}$ is the lth diagonal element of the measurement covariance matrix $\mathbf{R}_k$ in the simultaneous form of DEKF, that the scalar $\xi_{k,l}$ is the lth error signal, and that the vector $\mathbf{h}_{k,l}^i$ is the lth column of the augmented derivative matrix $\mathbf{H}_k^i$. After all output signals of all training streams have been processed, the weight vectors and error covariance matrices for all weight groups are updated by

$$\hat{\mathbf{w}}_{k+1}^i = \hat{\mathbf{w}}_k^i + \Delta\hat{\mathbf{w}}_{k,N_oN_s}^i, \tag{2.50}$$

$$\mathbf{P}_{k+1}^i = \mathbf{P}_{k,N_oN_s}^i + \mathbf{Q}_k^i. \tag{2.51}$$

Structurally, these equations for sequential updates are nearly identical to those of the simultaneous update, with the exception of an additional correction term in the delta-weight update equation; this term is necessary to account for the fact that the weight estimate changes at each step of this sequential multiple-output recursion.

2.6.4 Square-Root Filtering

2.6.4.1 Without Artificial Process Noise Sun and Marko [18] have described the use of square-root filtering as a numerically stable, alternative method to performing the approximate error covariance matrix update given by the Riccati equation (2.6). The square-root filter methods are well known in the signal processing community [19], and were developed so as to guarantee that the positive-definiteness of the matrix is maintained throughout training. However, this insurance is accompanied by increased computational complexity. Below, we summarize the square-root formulation for the case of no artificial process noise, with proper treatment of the EKF learning rate as given in Eq. (2.7) (we again assume $\mathbf{S}_k = \mathbf{I}$).

The square-root covariance filter update is based on the matrix factorization lemma, which states that for any pair of $J \times K$ matrices $\mathbf{B}_1$ and $\mathbf{B}_2$, with $J \leq K$, the relation $\mathbf{B}_1\mathbf{B}_1{}^T = \mathbf{B}_2\mathbf{B}_2{}^T$ holds if and only if there exists a unitary matrix Θ such that $\mathbf{B}_2 = \mathbf{B}_1\Theta$. With this in mind, the covariance update equations (2.3) and (2.6) can be written in matrix form as

$$\begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{H}_k^T\mathbf{P}_k^{1/2} \\ \mathbf{0} & \mathbf{P}_k^{1/2} \end{bmatrix} \begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{0} \\ \mathbf{P}_k^{1/2}\mathbf{H}_k & \mathbf{P}_k^{1/2} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_k^{-1/2} & \mathbf{0} \\ \mathbf{P}_k\mathbf{H}_k\mathbf{A}_k^{1/2} & \mathbf{P}_{k+1}^{1/2} \end{bmatrix} \begin{bmatrix} \mathbf{A}_k^{-1/2} & \mathbf{A}_k^{1/2}\mathbf{H}_k^T\mathbf{P}_k \\ \mathbf{0} & \mathbf{P}_{k+1}^{1/2} \end{bmatrix}. \tag{2.52}$$

Now, the idea is to find a unitary transformation Θ such that

$$\begin{bmatrix} \mathbf{A}_k^{-1/2} & \mathbf{0} \\ \mathbf{P}_k\mathbf{H}_k\mathbf{A}_k^{1/2} & \mathbf{P}_{k+1}^{1/2} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{H}_k^T\mathbf{P}_k^{1/2} \\ \mathbf{0} & \mathbf{P}_k^{1/2} \end{bmatrix} \Theta. \tag{2.53}$$

This is easily accomplished by applying a series of 2×2 Givens rotations to annihilate the elements of the submatrix $\mathbf{H}_k^T\mathbf{P}_k^{1/2}$, thereby yielding the left-hand-side matrix. Given this result of the square-root filtering procedure, we can perform the network weight update via the following additional steps: (1) compute $\mathbf{A}_k^{1/2}$ by inverting $\mathbf{A}_k^{-1/2}$; (2) compute the Kalman gain matrix by $\mathbf{K}_k = (\mathbf{P}_k\mathbf{H}_k\mathbf{A}_k^{1/2})\mathbf{A}_k^{1/2}$; (3) perform the weight update via Eq. (2.5).

2.6.4.2 With Artificial Noise In our original work [3] on EKF-based training, we introduced the use of artificial process noise as a simple and easily controlled mechanism to help assure that the approximate error covariance matrix $\mathbf{P}_k$ would retain the necessary property of nonnegative-definiteness, thereby allowing us to avoid the more computationally complicated square-root formulations. In addition to controlling the proper evolution of $\mathbf{P}_k$, we have also found that artificial process noise, when carefully applied, helps to accelerate the training process and, more importantly, leads to solutions superior to those found without artificial process noise. We emphasize that the use of artificial process noise is not ad hoc, but appears due to the process noise term in Eq. (2.1) (i.e., the covariance matrix $\mathbf{Q}_k$ in Eq. (2.6) disappears only when $\boldsymbol{\omega}_k = 0$ for all k). We have continued to use this feature in our implementations as an effective means for escaping poor local minima, and have not experienced problems with divergence. Other researchers with independent implementations of various forms of EKF [13, 17, 20] for neural network training have also found the use of artificial process noise to be beneficial. Furthermore, other gradient-based training algorithms have effectively exploited weight noise (e.g., see [21]).

We now demonstrate that the square-root filtering formulation of Eq. (2.53) is easily extended to include artificial process noise, and that the use of artificial process noise and square-root filtering are not mutually exclusive. Again, we wish to express the error covariance update in a

factorized form, but we now augment the left-hand-side of Eq. (2.52) to include the square root of the process-noise covariance matrix:

$$\begin{bmatrix} \mathbf{R}_k + \mathbf{H}_k^T \mathbf{P}_k \mathbf{H}_k & \mathbf{H}_k^T \mathbf{P}_k \\ \mathbf{P}_k \mathbf{H}_k & \mathbf{P}_k + \mathbf{Q}_k \end{bmatrix} = \begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{H}_k^T \mathbf{P}_k^{1/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_k^{1/2} & \mathbf{Q}_k^{1/2} \end{bmatrix} \begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{0} \\ \mathbf{P}_k^{1/2} \mathbf{H}_k & \mathbf{P}_k^{1/2} \\ \mathbf{0} & \mathbf{Q}_k^{1/2} \end{bmatrix}. \tag{2.54}$$

Similarly, the right-hand side of Eq. (2.52) is augmented by blocks of zeroes, so that the matrices are of the same size as those of the right-hand side of Eq. (2.54):

$$\begin{bmatrix} \mathbf{A}_k^{-1} & \mathbf{H}_k^T \mathbf{P}_k \\ \mathbf{P}_k \mathbf{H}_k & \mathbf{P}_{k+1} + \mathbf{P}_k \mathbf{H}_k \mathbf{A}_k \mathbf{H}_k^T \mathbf{P}_k \end{bmatrix} = \begin{bmatrix} \mathbf{A}_k^{-1/2} & \mathbf{0} & \mathbf{0} \\ \mathbf{P}_k \mathbf{H}_k \mathbf{A}_k^{1/2} & \mathbf{P}_{k+1}^{1/2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{A}_k^{1/2} & \mathbf{A}_k^{1/2} \mathbf{H}_k^T \mathbf{P}_k \\ \mathbf{0} & \mathbf{P}_{k+1}^{1/2} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{2.55}$$

Here, Eqs. (2.54) and (2.55) are equivalent to one another, and Eq. (2.53) is appropriately modified to

$$\begin{bmatrix} \mathbf{A}_k^{1/2} & \mathbf{0} & \mathbf{0} \\ \mathbf{P}_k \mathbf{H}_k \mathbf{A}_k^{1/2} & \mathbf{P}_{k+1}^{1/2} & \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_k^{1/2} & \mathbf{H}_k^T \mathbf{P}_k^{1/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_k^{1/2} & \mathbf{Q}_k^{1/2} \end{bmatrix} \Theta. \tag{2.56}$$

Thus, square-root filtering can easily accommodate the artificial process-noise extension, where the matrices $\mathbf{H}_k^T \mathbf{P}_k^{1/2}$ and $\mathbf{Q}_k^{1/2}$ are both annihilated via a sequence of Givens rotations. Note that this extension involves substantial additional computational costs beyond those incurred when $\mathbf{Q}_k = \mathbf{0}$. For a network with M weights and N_o outputs, the use of artificial process noise introduces $O(M^3)$ additional computations for the annihilation of $\mathbf{Q}_k^{1/2}$, whereas the annihilation of the matrix $\mathbf{H}_k^T \mathbf{P}_k^{1/2}$ only involves $\mathcal{O}(M^2 N_o)$ computations (here we assume $M \gg M_o$).

2.7 OTHER EXTENSIONS AND ENHANCEMENTS

2.7.1 EKF Training with Constrained Weights

Due to the second-order properties of the EKF training procedure, we have observed that, for certain problems, networks trained by EKF tend to develop large weight values (e.g., between 10 and 100 in magnitude). We view this capability as a double-edged sword: on the one hand, some problems may require that large weight values be developed, and the EKF procedures are effective at finding solutions for these problems. On the other hand, trained networks may need to be deployed with execution performed in fixed-point arithmetic, which requires that limits be imposed on the range of values of network inputs, outputs and weights. For nonlinear sigmoidal nodes, the node outputs are usually limited to values between -1 and $+1$, and input signals can usually be linearly transformed so that they fall within this range. On the other hand, the EKF procedures as described above place no limit on the weight values. We describe here a natural mechanism, imposed during training, that limits the range of weight values. In addition to allowing for fixed-point deployment of trained networks, this weight-limiting mechanism may also promote better network generalization.

We wish to set constraints on weight values during the training process while maintaining rigorous consistency with the EKF recursion. We can accomplish this by converting the unconstrained nonlinear optimization problem into one of optimization with constraints. The general idea is to treat each of the network's weight values as the output of a monotonically increasing function $\phi(\cdot)$ with saturating limits at the function's extremes (e.g., a sigmoid function). Thus, the EKF recursion is performed in an unconstrained space, while the network's weight values are nonlinear transformations of the corresponding unconstrained values that evolve during the training process. This transformation requires that the EKF recursion be modified to take into account the function $\phi(\cdot)$ as applied to the parameters (i.e., unconstrained weight values) that evolve during training.

Assume that the vector of network's weights $\mathbf{w}_k$ is constrained to take on values in the range $-\alpha$ to $+\alpha$ and that each component $w_k^{i,j}$ (the jth weight of the ith node) of the constrained weight vector is related to an unconstrained value $\tilde{w}_k^{i,j}$ via a function $w_k^{i,j} = \phi(\tilde{w}_k^{i,j}, \alpha)$. We formulate the EKF recursion so that weight updates are performed in the unconstrained weight space, while the steps of forward propagation and backpropagation of derivatives are performed in the constrained weight space.

The training steps are carried out as follows. At time step k, an input vector is propagated through the network, and the network outputs are computed and stored in the vector $\hat{\mathbf{y}}_k$. The error vector $\boldsymbol{\xi}_k$ is also formed as defined above. Subsequently, the derivatives of each component of $\hat{\mathbf{y}}_k$ with respect to each node's weight vector $\mathbf{w}_k^i$ are computed and stored into the matrices $\mathbf{H}_k^i$, where the component $\mathbf{H}_k^{i,j,l}$ contains the derivative of the lth component of $\hat{\mathbf{y}}_k$ with respect to the jth weight of the ith node. In order to perform the EKF recursion in the unconstrained space, we must perform three steps in addition to those that are normally carried out. First, we transform weight values from the constrained space via $\tilde{w}_k^{i,j} = \phi^{-1}(w_k^{i,j}, \alpha)$ for all trainable weights of all nodes of the network, which yields the vectors $\tilde{\mathbf{w}}_k^i$. Second, the derivatives that have been previously computed with respect to weights in the constrained space must be transformed to derivatives with respect to weights in the unconstrained space. This is easily performed by the following transformation for each derivative component:

$$\tilde{H}_k^{i,j,l} = H_k^{i,j,l} \frac{\partial \omega_k^{i,j}}{\partial \tilde{w}_k^{i,j}} = H_k^{i,j,l} \frac{\partial \phi(\tilde{w}_k^{i,j}, \alpha)}{\partial \tilde{w}_k^{i,j}}. \tag{2.57}$$

The EKF weight update procedure of Eqs. (2.8)–(2.11) is then applied using the unconstrained weights and derivatives with respect to unconstrained weights. Note that no transformation is applied to either the scaling matrix $\mathbf{S}_k$ or the error vector $\boldsymbol{\xi}_k$ before they are used in the update. Finally, after the weight updates are performed, the unconstrained weights are transformed back to the constrained space by $w_k^{i,j} = \phi(\tilde{w}_k^{i,j}, \alpha)$ for all weights of all nodes in the network.

We now consider specific forms for the function $\phi(\tilde{w}_k^{i,j}, \alpha)$ that transforms weight values from an unconstrained space to a constrained space. We require that the function obey the following properties:

1. $\phi(\tilde{w}_k^{i,j}, \alpha)$ is monotonically increasing.
2. $\phi(0, \alpha) = 0$.
3. $\phi(-\infty, \alpha) = -\alpha$.
4. $\phi(+\infty, \alpha) = +\alpha$.
5. $\dfrac{\partial w_k^{i,j}}{\partial \tilde{w}_k^{i,j}}\Big|_{\tilde{w}_k^{i,j}=0} = 1$.
6. $\lim_{\alpha \to \infty} \phi(\tilde{w}_k^{i,j}, \alpha) = \tilde{w}_k^{i,j} = w_k^{i,j}$.

Property 5 imposes a constraint on the gain of the transformation, while property 6 imposes the constraint that in the limit of large α, the constrained optimization problem operates identically to the unconstrained problem. This last constraint also implies that $\lim_{\alpha\to\infty}(\partial w_k^{i,j}/\partial\tilde{w}_k^{i,j}) = 1$.

One candidate function is a symmetric saturating linear transformation where $w_k^{i,j} = \tilde{w}_k^{i,j}$ when $-\alpha \le \tilde{w}_k^{i,j} \le \alpha$, and is otherwise equal to either saturating value. The major disadvantage with this constraint function is that its inverse is multivalued outside the linear range. Thus, once the training process pushes the constrained weight value into the saturated region, the derivative of constrained weight with respect to unconstrained weight becomes zero, and no further training of that particular weight value will occur due to the zero-valued derivative.

Alternatively, we may consider various forms of symmetric sigmoid functions that are everywhere differentiable and have well-defined inverses. The Elliott sigmoid [22] conveniently does not involve transcendental functions. We choose to consider a generalization of this monotonic and symmetric saturating function given by

$$w_k^{i,j} = \phi(\tilde{w}_k^{i,j}, \alpha) = \frac{\alpha\tilde{w}_k^{i,j}}{\beta + |\tilde{w}_k^{i,j}|} = \frac{\tilde{w}_k^{i,j}}{\beta/\alpha + |\tilde{w}_k^{i,j}|/\alpha}, \tag{2.58}$$

where β is a positive quantity that determines the function's gain. We must choose the value of β so that the derivative of $\phi(\tilde{w}_k^{i,j}, \alpha)$ with respect to $\tilde{w}_k^{i,j}$, evaluated at $\tilde{w}_k^{i,j} = 0$, is equal to 1. This condition can be shown to be satisfied by the choice $\beta = \alpha$. Thus, the constraint function we choose is given by

$$w_k^{i,j} = \phi(\tilde{w}_k^{i,j}, \alpha) = \frac{\alpha\tilde{w}_k^{i,j}}{\alpha + |\tilde{w}_k^{i,j}|} = \frac{\tilde{w}_k^{i,j}}{1 + |\tilde{w}_k^{i,j}|/\alpha}. \tag{2.59}$$

By inspection, we see that this function satisfies the various requirements. For example, for large α (i.e., as $\alpha \to \infty$), $w_k^{i,j} \to \tilde{w}_k^{i,j}$; for smaller values of α, when $|\tilde{w}_k^{i,j}| \gg \alpha$, $w_k^{i,j} \to \alpha\ \mathrm{sgn}(\tilde{w}_k^{i,j})$. The inverse of this function is easily found to be given by

$$\tilde{w}_k^{i,j} = \phi^{-1}(w_k^{i,j}, \alpha) = \frac{\alpha w_k^{i,j}}{\alpha - |w_k^{i,j}|} = \frac{w_k^{i,j}}{1 - |w_k^{i,j}|/\alpha}. \tag{2.60}$$

In this case, for $\alpha \gg w_k^{i,j}$, $\tilde{w}_k^{i,j} \to w_k^{i,j}$; similarly, as $w_k^{i,j} \to \alpha$, $\tilde{w}_k^{i,j} \to \infty$. As a final note, the derivative of constrained weight with respect to uncon-

strained weight, which is needed for computing the proper derivatives in the EKF recursion, can be expressed in many different ways, some of which are given by

$$\frac{\partial w_k^{i,j}}{\partial \tilde{w}_k^{i,j}} = \left(\frac{\alpha}{\alpha + |\tilde{w}_k^{i,j}|}\right)^2 = \left(\frac{\alpha - |w_k^{i,j}|}{\alpha}\right)^2 = \left(1 - \frac{|w_k^{i,j}|}{\alpha}\right)^2 = \left(\frac{w_k^{i,j}}{\tilde{w}_k^{i,j}}\right)^2. \quad (2.61)$$

2.7.2 EKF Training with an Entropic Cost Function

As defined above, the EKF training algorithm assumes that a quadratic function of some error signal is being minimized over all network outputs and all training patterns. However, other cost functions are often useful or necessary. One such function that has been found to be particularly appropriate for pattern classification problems, and for which a sound statistical basis exists, is a cost function based on minimizing cross-entropy [23]. We consider a prototypical problem in which a network is trained to act as a pattern classifier; here network outputs encode binary pattern classifications. We assume that target values of ± 1 are provided for each training pattern. Then the contribution to the total entropic cost function at time step n is given by

$$\mathscr{E}_k = \sum_{l=1}^{N_o} \mathscr{E}_k^l = \sum_{l=1}^{N_o} \left[(1 + y_k^l) \log \frac{1 + y_k^l}{1 + \hat{y}_k^l} + (1 - y_k^l) \log \frac{1 - y_k^l}{1 - \hat{y}_k^l} \right]. \quad (2.62)$$

Since the components of the vector $\mathbf{y}_k$ are constrained to be either $+1$ or -1, we note that only one of the two components for each output l will be nonzero. This allows the cost function to be expressed as

$$\mathscr{E}_k|_{y_k^l = \pm 1} = \sum_{l=1}^{N_o} \mathscr{E}_k^l = \sum_{l=1}^{N_o} 2 \log \frac{2}{1 + y_k^l \hat{y}_k^l}. \quad (2.63)$$

The EKF training procedure assumes that at each time step k a quadratic cost function is being minimized, which we write as $C_k = \sum_{l=1}^{N_o} (z_k^l - \hat{z}_k^l)^2 = \sum_{l=1}^{N_o} (\xi_k^l)^2$, where z_k^l and $\hat{z}_k^l$ are target and output values, respectively. (We assume here the case of $\mathbf{S}_k = \mathbf{I}$; this procedure is easily extended to nonuniform weighting matrices.) At this point, we would like to find appropriate transformations between the $\{z_k^l, \hat{z}_k^l\}$ and $\{y_k^l, \hat{y}_k^l\}$ so that C_k and $\mathscr{E}_k$ are equivalent, thereby allowing us

to use the EKF procedure to minimize the entropic cost function. We first note that both the quadratic and entropic cost functions are calculated by summing individual cost function components from all targets and output nodes. We immediately see that this leads to the equality $(\xi_k^l)^2 = \varepsilon_k^l$ for all N_o outputs, which implies $z_k^l - \hat{z}_k^l = (\varepsilon_k^l)^{1/2}$. At this point, we assume that we can assign all target values $z_k^l = 0$,[3] so that $\xi_k^l = -\hat{z}_k^l = (\varepsilon_k^l)^{1/2}$. Now the EKF recursion can be applied to minimize the entropic cost function, since $C^k = \Sigma_{l=1}^{N_o}(\xi_k^l)^2 = \Sigma_{l=1}^{N_o}[(\varepsilon_k^l)^{1/2}]^2 = \Sigma_{l=1}^{N_o}\varepsilon_k^l = \mathscr{E}_k$.

The remainder of the derivation is straightforward. The EKF recursion in the case of the entropic cost function requires that derivatives of $\hat{z}_k^l = (-\varepsilon_k^l)^{1/2}$ be computed for all N_o outputs and all weight parameters, which are subsequently stored in the matrices $\mathbf{H}_k^i$. Applying the chain rule, these derivatives are expressed as a function of the derivatives of network outputs with respect to weight parameters:

$$H_k^{i,j,l} = \left.\frac{\partial \hat{z}_k^l}{\partial w_k^{i,j}}\right|_{y_k^l = \pm 1} = -\frac{\partial(\varepsilon_k^l)^{1/2}}{\partial w_k^{i,j}} = \frac{1}{(\varepsilon_k^l)^{1/2}(y_k^l + \hat{y}_k^l)}\frac{\partial \hat{y}_k^l}{\partial w_k^{i,j}}. \tag{2.64}$$

Note that the effect of the relative entropy cost function on the calculation of derivatives is handled entirely in the initialization of the backpropagation process, where the term $1/[(\varepsilon_k^l)^{1/2}(y_k^l + \hat{y}_k^l)]$ is used for each of the N_o output nodes to start the backpropagation process, rather than starting with a value of unity for each output node as in the nominal formulation.

In general, the EKF procedure can be modified in the manner just described for a wide range of cost functions, provided that they meet at least three simple requirements. First, the cost function must be a differentiable function of network outputs. Second, the cost function should be expressed as a sum of contributions, where there is a separate target value for each individual component. Third, each component of the cost function must be non-negative.

2.7.3 EKF Training with Scalar Errors

When applied to a multiple-output training problem, the EKF formulation in Eqs. (2.3)–(2.6) requires a separate backpropagation for each output and a matrix inversion. In this section, we describe an approximation to

[3]The idea of using a modified target value of zero with the actual targets appearing in expressions for system outputs can be applied to the EKF formulation of Eqs. (2.3)–(2.6) without any change in its underlying behavior.

the EKF neural network training procedure that allows us to treat such problems with single-output training complexity. In this approximation, we require only the computation of derivatives of a scalar quantity with respect to trainable weights, thereby reducing the backpropagation computation and eliminating the need for a matrix inversion in the multiple-output EKF recursion.

For the sake of simplicity, we consider here the prototypical network training problem for which network outputs directly encode signals for which targets are defined. The square root of the contribution to the total cost function at time step k is given by

$$\tilde{y}_k = C_k^{1/2} = \left(\sum_{l=1}^{N_o} |y_k^l - \hat{y}_k^l|^2 \right)^{1/2}, \tag{2.65}$$

where we are again treating the simple case of uniform scaling of network errors (i.e., $\mathbf{S}_k = \mathbf{I}$). The goal here is to train a network so that the sum of squares of this scalar error measure is minimized over time. As in the case of the entropic cost function, we consider the target for training to be zero for all training instances, and the scalar error signal used in the Kalman recursion to be given by $\xi_k = 0 - \tilde{y}_k$. The EKF recursion requires that the derivatives of the scalar observation $\tilde{y}_k$ be computed with respect to all weight parameters. The derivative of the scalar error with respect to the jth weight of the ith node is given by

$$H_k^{i,j,1} = \frac{\partial \tilde{y}_k}{\partial w_k^{i,j}} = \sum_{l=1}^{N_o} \frac{\partial \tilde{y}_k}{\partial \hat{y}_k^l} \frac{\partial \hat{y}_k^l}{\partial w_k^{i,j}} = \sum_{l=1}^{N_o} \frac{y_k^l - \hat{y}_k^l}{\xi_k} \frac{\partial \hat{y}_k^l}{\partial w_k^{i,j}}. \tag{2.66}$$

In this scalar formulation, the derivative calculations via backpropagation are initialized with the terms $(y_k^l - \hat{y}_k^l)/\xi_k$ for all N_o network output nodes (as opposed to initializing the backpropagation calculations with values of unity for the nominal EKF recursion of Eqs. (2.3)–(2.6)). Furthermore, only one quantity is backpropagated, rather than N_o quantities for the nominal formulation. Note that this scalar approximation reduces exactly to the nominal EKF algorithm in the limit of a single-output problem:

$$\tilde{y}_k = (|y_k^1 - \hat{y}_k^1|^2)^{1/2} = |y_k^1 - \hat{y}_k^1|, \tag{2.67}$$

$$\xi_k = 0 - |y_k^1 - \hat{y}_k^1|, \tag{2.68}$$

$$H_k^{i,j,1} = \frac{\partial \tilde{y}_k}{\partial w_k^{i,j}} = -\text{sgn}(y_k^1 - \hat{y}_k^1) \frac{\partial \hat{y}_k^1}{\partial w_k^{i,j}}. \tag{2.69}$$

Consider the case $y_k^1 \leq \hat{y}_k^1$. Then, the error signal is given by $\xi_k = y_k^1 - \hat{y}_k^1$. Similarly, $\partial\tilde{y}_k/\partial w = \partial\hat{y}_k^1/\partial w$, since $\partial\tilde{y}_k/\partial y_k^1 = 1$. Otherwise, when $y_k^1 > \hat{y}_k^1$, the error signal is given by $\xi_k = -(y_k^1 - \hat{y}_k^1)$, and $\partial\tilde{y}_k/\partial w = -\partial\hat{y}_k^1/\partial w$, since $\partial\tilde{y}_k/\partial\hat{y}_k^1 = -1$. Since both the error and the derivatives are the negatives of what the nominal EKF recursion provides, the effects of negation cancel one another. Thus, in either case, the scalar formulation for a single-output problem is exactly equivalent to that of the EKF procedure of Eqs. (2.3)–(2.6).

Because the procedure described here is an approximation to the base procedure, we suspect that classes of problems exist for which it is not as effective; further work will be required to clarify this question. In this regard, we note that once criteria are available to guide the decision of whether to scalarize or not, one may also consider a hybrid approach to problems with many outputs. In this approach, selected outputs would be combined as described above to produce scalar error variables; the latter would then be treated with the original procedure.

2.8 AUTOMOTIVE APPLICATIONS OF EKF TRAINING

The general area of automotive powertrain control, diagnosis, and modeling has offered substantial opportunity for the application of neural network methods. These opportunities 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. We summarize the application of EKF training to three signal processing problems related to automotive diagnostics and emissions modeling, as well as its application to two automotive control problems. In all five cases, we have found EKF training of recurrent neural networks to be an enabler for developing effective solutions to these problems.

Figure 2.7 provides a diagrammatic representation of these five neural network applications and how they potentially interact with one another. We observe that the neural network controllers for engine idle speed and air/fuel (A/F) ratio control produce signals that affect the operation of the engine, while the remaining neural network models are used to describe various aspects of engine operation as a function of measurable engine outputs.

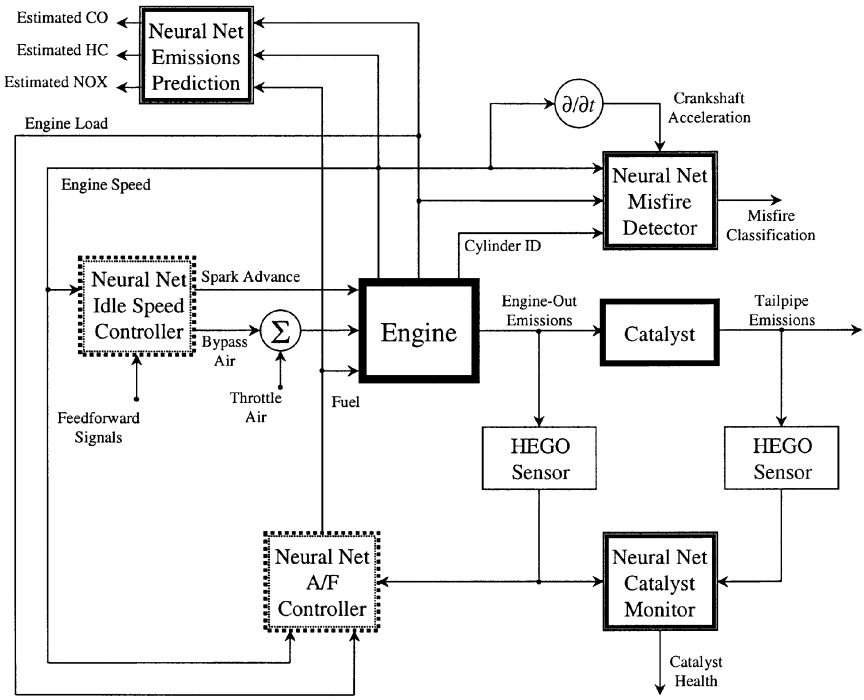

Figure 2.7 Block-diagram representation of neural network applications for automotive engine control and diagnosis. Solid boxes represent physical components of the engine system, double-lined solid boxes represent neural network models or diagnostic processes, and double-lined dashed boxes represent neural network controllers. For the sake of simplicity, we have not shown all relevant sensors and their corresponding signals (e.g., engine coolant temperature).

2.8.1 Air/Fuel Ratio Control

At a very basic level, the role of the A/F controller is to supply fuel to the engine such that it matches the amount of air pumped into the engine via the throttle and idle speed bypass valve. 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, as well as engine operating conditions such as engine speed and load, 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 rich/lean 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. We have demonstrated that an open-loop recurrent neural network controller can be trained to provide a correction signal to the closed-loop A/F control in the face of transient conditions (i.e., dynamic changes in engine speed and load), thereby eliminating large deviations from stoichiometry. This is accomplished by using an auxiliary universal EGO (UEGO) sensor, which provides a continuous measure of A/F ratio (as opposed to the rich/lean indication provided by the HEGO), during the in-vehicle training process. Deviations of measured A/F ratio from stoichiometric A/F ratio provide the error signal for the EKF training process; however, the measured A/F ratio is not used as an input, and since the A/F control does not have a major effect on engine operating conditions when operated near stoichiometry, then this can be viewed as a problem of training an open-loop controller. Nevertheless, we use recurrent network controllers to provide the capability of representing the condition-dependent dynamics associated with the operation of the engine system under A/F control, and must take care to properly compute derivatives with BPTT(h).

2.8.2 Idle Speed Control

A second engine control task is that of maintaining smooth engine operation at idle conditions. In this case, no air is provided to the intake manifold of the engine via the throttle; in order to keep the engine running, a bypass air valve is used to regulate the flow of air into the engine. The role of the idle speed control system is to maintain a relatively low (for purposes of fuel economy) and constant engine speed, in the face of disturbances that place and remove additional loads on the engine (e.g., shifting from neutral to drive, activating the air conditioning system, and locking up the power steering); feedforward signals encoding these events are provided as input to the idle speed controller. The control range of the bypass air signal is large (more than 1000 rpm under idle conditions), but its effect is delayed by a time inversely proportional to engine speed. The spark advance command, which regulates the timing of ignition, has an immediate effect on engine speed, but over a small range (on order of 100 rpm). Thus, an effective engine idle speed controller coordinates the two controls to maintain a constant engine speed. The error signals for the EKF training process are a weighted sum of squared deviations of engine

speed from a desired speed, combined with constraints on the controls expressed as squared error signals. We have used recurrent neural networks, trained by on-line EKF methods, to develop effective idle speed control strategies, and have documented this work in [5]. Note that unlike the case of the A/F controller, this is an example of a closed-loop controller, since the bypass air and spark advance controls affect engine speed, which is used as a controller input.

2.8.3 Sensor-Catalyst Modeling

A particularly critical component of a vehicle's emissions control system is the catalytic converter. The role of the catalytic converter is to chemically transform noxious and environmentally damaging engine-out emissions, which are the byproduct of the engine's combustion process, to environmentally benign chemical compounds. An ideal three-way 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 CO_2; and (3) reduction of nitrogen oxides (NO_x) to nitrogen (N_2) and oxygen (O_2). In practice, it is possible to achieve high conversion efficiencies for all three types of exhaust gases only when the engine is operating near stoichiometry. An effective A/F control strategy enables such conversion.

However, even in the presence of effective A/F control, vehicle-out (i.e., tailpipe) emissions may be unreasonably high if the catalytic converter has been damaged. 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, catalytic converter monitors are based on comparing the output of a HEGO sensor that is exposed to engine-out emissions with the output of a second sensor that is mounted downstream of the catalytic converter and is exposed to the tailpipe emissions. This approach is based on the observation that the postcatalyst HEGO sensor switches infrequently, relative to the precatalyst HEGO sensor, when the catalyst is operating efficiently. Similarly, the average rate of switching of the postcatalyst sensor increases as catalyst efficiency decreases (due to decreasing oxygen storage capability).

A catalyst monitor can be developed based on a neural network model of the dynamic operation of the postcatalyst HEGO sensor as a function of

the precatalyst HEGO sensor and engine operating conditions [12] for a catalyst of nominal conversion efficiency. This is a difficult task, especially given the nonlinear responses of the various components and the condition-dependent time delays, which can range from less than 0.1 s at high engine speeds to more than 1 s at low speeds. We employed a RMLP network with structure 15-20R-15R-10R-1 and a sparse tapped delay line representation to directly capture the long-term temporal characteristics of the precatalyst HEGO sensor. Because of the size of the network (over 1,500 weights) and the number of training samples (63,000), we chose to employ decoupled EKF training. The trained network effectively represented the condition-dependent time delays and nonlinearities of the system, as shown in [12].

2.8.4 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 misfire will lead to a deterioration of the catalytic converter, ultimately resulting in unacceptable levels of emitted pollutants. Consequently, government mandates require that onboard misfire detection capability be provided for nearly all engine operating conditions.

While there are many ways of detecting engine misfire, all currently practical methods rely on observing engine crankshaft 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.

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, from 20 ms at 750 rpm to 2.5 ms at 6000 rpm for an eight-cylinder engine. 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.

A serious problem associated with measuring crankshaft acceleration is the presence of complex torsional dynamics of the crankshaft, even in the absence of misfire. This is due to the finite stiffness of the crankshaft. 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 followed by misfire may be misinterpreted.

We have approached the misfire detection problem with recurrent neural networks trained by GEKF [12] to act as dynamic pattern classifiers. We use as inputs engine speed, engine load, crankshaft acceleration, and a binary flag to identify the beginning of the cylinder firing sequence. The training target is a binary signal, according to whether a misfire had been artificially induced for the current cylinder during the previous engine cycle. This phasing enables the network to make use of information contained in measured accelerations that follow the engine event being classified. We find that trained networks make remarkably few classification errors, most of which occur during moments of rapid acceleration or deceleration.

2.8.5 Vehicle Emissions Estimation

Increasing levels of pollutants in the atmosphere – observed despite the imposition of stricter emission standards and technological improvements in emissions control systems – have led to models being developed to predict emissions inventories. These are typically based on the emissions levels that are mandated by the government for a particular driving schedule and a given model year. It has been found that the emissions inventories based on these mandated levels do not accurately reflect those that are actually found to exist. That is, actual emission rates depend heavily upon driving patterns, and real-world driving patterns are not comprehensively represented by the mandated driving schedules. To better assess the emissions that occur in practice and to predict emissions inventories, experiments have been conducted using instrumented vehicles that are driven in actual traffic. Unfortunately, such vehicles are costly and are difficult to operate and maintain.

We have found that recurrent neural networks can be trained to estimate instantaneous engine-out emissions from a small number of easily measured engine variables. Under the assumption of a properly operating fuel control system and catalytic converter, this leads to estimates of tailpipe emissions as well. This capability then allows one to estimate the sensitivity of emissions to driving style (e.g., aggressive versus conservative). Once trained, the network requires only information already available to the powertrain processor. Because of engine dynamics, we have found the use of recurrent networks trained by EKF methods to enable

accurate estimation of instantaneous emissions levels. We provide a detailed description of this application in [24].

2.9 DISCUSSION

We have presented in this chapter an overview of neural network training methods based on the principles of extended Kalman filtering. We summarize our findings by considering the virtues and limitations of these methods, and provide guidelines for implementation.

2.9.1 Virtues of EKF Training

The EKF family of training algorithms develops and employs second-order information during the training process using only first-order approximations. The use of second-order information, as embedded in the approximate error covariance matrix, which co-evolves with the weight vector during training, provides enhanced capabilities relative to first-order methods, both in terms of training speed and quality of solution. The amount of second-order information utilized is controlled by the level of decoupling, which is chosen on the basis of computational considerations. Thus, the computational complexity of the EKF methods can be scaled to meet the needs of specific applications.

We have found that EKF methods have enabled the training of recurrent neural networks, for both modeling and control of nonlinear dynamical systems. The sequential nature of the EKF provides advantages relative to batch second-order methods, since weight updates can be performed on an instance-by-instance basis with EKF training. On the other hand, the ability to batch multiple training instances with multistream EKF training provides a level of scalability in addition to that provided by decoupling. The sequential nature of the EKF, in both single- and multistream operation, provides a stochastic component that allows for more effective search of the weight space, especially when used in combination with artificial process noise.

The EKF methods are easily implemented in software, and there is substantial promise for hardware implementation as well. Methods for avoiding matrix inversions in the EKF have been developed, thereby enabling easy implementations. Finally, we believe that the greatest virtue of EKF training of neural networks is its established and proven applicability to a wide range of difficult modeling and control problems.

2.9.2 Limitations of EKF Training

Perhaps the most significant limitation of EKF training is its limited applicability to cost functions other then minimizing sum of squared error. Although we have shown that other cost functions can be used (e.g. entropic measures), we are nevertheless restricted to those optimization problems that can be converted to minimizing a sum of squared error criterion. On the other hand, many problems, particularly in control, require other optimization criteria. For example, in a portfolio optimization problem, we should like to *maximize* the total return over time. Converting such an optimization criterion to a sum of squared errors criterion is usually not straightforward. However, we do not view the sum of squared-error optimization criterion as a limitation for most problems that can be viewed as belonging to the class of traditional supervised training problems.

The EKF procedures described in this chapter are derived on the basis of a first-order linearization of the nonlinear system; this may provide a limitation in the form of large errors in the weight estimates and covariance matrix, since the second-order information is effectively developed by taking outer products of the gradients. Chapter 7 introduces the unscented Kalman filter (UKF) as an alternative to the EKF. The UKF is expected to provide a more accurate means of developing the required second-order information than the EKF, without increasing the computational complexity.

2.9.3 Guidelines for Implementation and Use

1. Decoupling should be used when computation is a concern (e.g., for on-line applications). Node and layer decoupling are the two most appropriate choices. Otherwise, we recommend the use of global EKF, regardless of network architecture, as it should be expected to find better solutions than any of the decoupled versions because of the use of full second-order information.
2. Effectively, two parameter values need to be chosen for training of networks with EKF methods. We assume that the approximate error covariance matrices are always initialized with diagonal value of 100 and 1,000 for weights corresponding to nonlinear and linear nodes, respectively. Then, the user of these methods must set values for the learning rate and process-noise term according to characteristics of the training problem.

3. Training of recurrent networks, either as supervised training tasks or for controller training, can often be improved by multistreaming. The choice of the number of streams is dictated by problem characteristics.
4. Matrix inversions can be avoided by use of sequential EKF update procedures. In the case of decoupling, the order in which outputs are processed can affect training performance in detail. We recommend that outputs be processed in random order when these methods are used.
5. Square-root filtering can be employed to insure computational stability for the error covariance update equation. However, the use of square-root filtering with artificial process noise for covariance updates results in a substantial increase in computational complexity. We have noted that nonzero artificial process noise benefits training, by providing a mechanism to escape poor local minima and a mechanism that maintains stable covariance updates when using the Riccati update equation. We recommend that square-root filtering only be employed when no artificial process noise is used (and only for GEKF).
6. The EKF procedures can be modified to allow for alternative cost functions (e.g., entropic cost functions) and for weight constraints to be imposed during training, which thereby allow networks to be deployed in fixed-point arithmetic.

REFERENCES

[1] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning representations of back-propagation errors," *Nature* **323**, 533–536 (1986).

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

[3] G.V. Puskorius and L. A. Feldkamp, "Decoupled extended Kalman filter training of feedforward layered networks," in *Proceedings of International Joint Conference of Neural Networks*, Seattle, WA, 1991, Vol. 1, pp. 771–777.

[4] G.V. Puskorius and L.A. Feldkamp, "Automotive engine idle speed control with recurrent neural networks," in *Proceedings of the 1993 American Control Conference*, San Francisco, CA, pp. 311–316.

[5] 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*, **84**, 1407–1420 (1996).

[6] R.J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, **1**, 270–280 (1989).

[7] G.V. Puskorius and L.A. Feldkamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter-trained recurrent networks," *IEEE Transactions on Neural Networks*, **5**, 279–297 (1994).

[8] P.J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, **78**, 1550–1560 (1990).

[9] G.V. Puskorius and L.A. Feldkamp, "Extensions and enhancements of decoupled extended Kalman filter training," in *Proceedings of the 1997 International Conference on Neural Networks*, Houston, TX, Vol. 3, pp. 1879–1883.

[10] L.A. Feldkamp and G.V. Puskorius, "Training controllers for robustness: multi-stream DEKF," in *Proceedings of the IEEE International Conference on Neural Networks*, Orlando, FL, 1994, Vol. IV, pp. 2377–2382.

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

[12] 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*, **86**, 2259–2277 (1998).

[13] F. Heimes, "Extended Kalman filter neural network training: experimental results and algorithm improvements," in *Proceedings of the 1998 IEEE Conference on Systems, Man and Cybernetics*, Orlando, FL., pp. 1639–1644.

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

[15] G.H. Golub and C.F. Van Loan, *Matrix Computations*, 2nd ed. Baltimore, MD: The John Hopkins University Press, 1989.

[16] G.V. Puskorius and L.A. Feldkamp, "Avoiding matrix inversions for the decoupled extended Kalman filter training algorithm," in *Proceedings of the World Congress on Neutral Networks*, Washington, DC, 1995, pp. I-704–I-709.

[17] E.S. Plumer, "Training neural networks using sequential extended Kalman filtering," in *Proceedings of the World Congress on Neural Networks*, Washington DC, 1995 pp. I-764–I-769.

[18] P. Sun and K. Marko, "The square root Kalman filter training of recurrent neural networks," in *Proceedings of the 1998 IEEE Conference on Systems, Man and Cybernetics*, Orlando, FL, pp. 1645–1651.

[19] S. Haykin, *Adaptive Filter Theory*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall.

[20] E.W. Saad, D.V. Prokhorov, and D.C. Wunsch III, "Comparative study of stock trend prediction using time delay, recurrent and probabilistic neural networks," *IEEE Transactions on Neural Networks*, **9**, 1456–1470 (1998).

[21] K.-C. Jim, C.L. Giles, and B.G. Horne, "An analysis of noise in recurrent neural networks: convergence and generalization," *IEEE Transactions on Neural Networks*, **7**, 1424–1438 (1996).

[22] D.L. Elliot, "A better activation function for artificial neural networks," Institute for Systems Research, University of Maryland, Technical Report TR93-8, 1993.

[23] J. Hertz, A. Krogh, and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Redwood City, CA: Addison-Wesley, 1991.

[24] G. Jesion, C.A. Gierczak, G.V. Puskorius, L.A. Feldkamp, and J.W. Butler, "The application of dynamic neural networks to the estimation of feedgas vehicle emissions," in *Proceedings of the 1998 International Joint Conference on Neural Networks*, Anchorage, AK, pp. 69–73.