Chapter 1

Dynamical Recurrent Networks in Control

Danil V. Prokhorov, Gintaras V. Puskorius, and Lee A. Feldkamp

1.1 Introduction

We discuss in this chapter the application of dynamical neural networks to control. We describe our approach to training of such networks and summarize our experience accumulated through extensive computer simulations and experimentation with time-lagged recurrent networks used as components of a control system. In the past we have experimented with both real control systems and simulations thereof. In this chapter, for the sake of reproducibility, we adopt the setting of neurocontrol systems developed and tested in simulation. We provide two examples to illustrate our approach.

Simulation studies offer ideal opportunities to shape our understanding and command of neural network techniques. This philosophy is particularly appropriate for understanding dynamical neural networks. We restrict attention to time-lagged recurrent neural networks (TLRNN), with recurrent multilayer perceptrons (RMLP) as a special case. TLRNN's admit an arbitrary pattern of connectivity between nodes of a network, whereas RMLP's have a layered connectivity pattern. From a structural point of view, TLRNN's can subsume many conventional signal processing structures, e.g., tapped delay lines, finite impulse response (FIR) and infinite impulse response (IIR) filters. At the same time, TLRNN's can represent dynamic systems with strongly hidden states and possess universal approximation capabilities for dynamic systems similar to the capabilities of feedforward neural networks used for static mapping [Lo, 1993]. We suspect that internal recurrence may be more effective than output-to-input (external) recurrence. It seems preferable to place the recurrence closer to where it occurs in the dynamic system under control (*plant* in control parlance), e.g.,

before the output transform. In addition, internal states are not forced to be equal to some specific values. This tends to make the network better at iterated predictions than a similarly trained network with external recurrence only. Furthermore, we have shown that a properly trained fixed-weight TLRNN can act as a stabilizing controller for multiple distinct and weakly related systems, without explicit knowledge of system identity. We consider this last observation as particularly intriguing since it suggests that such a TLRNN effectively embeds characteristics of an *adaptive* system.

Recurrent networks are often regarded as difficult to train. Although our effective training procedures for TLRNN can be carried out with relative ease, we agree that difficulties observed in training are real. Common pitfalls are poor local minima and the recency effect, the tendency for recent weight updates to cause a network to forget what it has learned in the past. It is a reflection of the stability-plasticity dilemma faced by any neural network [Grossberg, 1982]. While the recency effect is also present in training feedforward networks, it is substantially more difficult to counter in training TLRNN's, where the temporal order of data sequences must be preserved. We nevertheless employ sequential training procedures for weight updates (pattern-by-pattern, rather than batch, updates) because of benefits associated with their stochastic nature. To avoid the recency effect, we have developed a special technique called multi-streaming. In essence, a single multi-stream update provides a "quasi-batching" mechanism that allows the weight update procedure to synthesize properly a composite update based on two or more instances. We emphasize that the resulting update is not a simple average of individual updates, and we discuss the details of multi-streaming in Section 1.4.3.

This chapter consists of the following sections. Section 1.2 introduces equations necessary for the execution of a TLRNN (forward propagation of signals). Section 1.3 describes elements of training, composed of derivative calculations and weight update methods. We briefly discuss two methods of calculating derivatives in TLRNN and emphasize the importance of obtaining meaningful derivatives. One method is called real-time recurrent learning (RTRL), the other backpropagation through time (BPTT) or truncated BPTT, denoted as BPTT(h). We prefer BPTT(h) because it is simpler to implement and requires much less computational effort than RTRL. Although we mention two weight update methods, first-order (such as gradient descent) and second-order, we use sequential second-order weight updates based on the extended Kalman filter (EKF) algorithm. Supplemented by multi-streaming (see Section 1.4.3), this method is one of the best modern methods of weight updates for TLRNN's.

We describe our basic approach to controller synthesis in Section 1.4. First, we discuss the specific elements of controller training. We show how recurrence originates naturally in any closed-loop control system, even if all of its components are feedforward neural networks. In general, this demands using more than one-time-step BPTT for best results. We mention the advantages of a recurrent neurocontroller, some of which are illustrated in Section 1.5. We adopt a modular approach that relies on a set of well tested functions (we use the C language, though other programming languages can certainly be used).

Each of the functions represents a particular procedure required to carry out a typical controller training session. For example, we have a separate function to propagate signals forward through a network given its description (architectural information and weights). Another function handles derivative calculations and BPTT. Yet another is used for EKF-based weight updates. We "wire" the required functions together for a specific problem of controller synthesis, which begins with development of a plant model. For a plant model, we can use either the generating equations directly or train an identification neural network. In either case, we may obtain the sensitivity signals that are required to train a controller. We conclude Section 1.4 by discussing our multi-stream technique.

Examples of controller training for nontrivial systems are given in Sections 1.5 and 1.6, followed by concluding remarks in Section 1.7.

1.2 Description and Execution of TLRNN

Let the TLRNN (Figure 1.1) consist of `n_nodes` nodes, including `n_in` nodes which serve as receptors for the external inputs, but not including the bias input, which we denote formally as node 0. The latter is set to the constant 1.0. The array $\mathbf{I}$ contains a list of the input nodes; e.g., I_j is the number of the node that corresponds to the `j`th input, in_j. Similarly, a list of the nodes that correspond to network outputs out_p is contained in the array $\mathbf{O}$. We allow for the possibility of network outputs and targets to be advanced or delayed with respect to node outputs by assigning a phase τ_p to each output. In most cases these phases are zero. Node i receives input from `n_con(i)` other nodes and has activation function $f_i(\cdot)$; `n_con(i)` is zero if node `i` is among the nodes listed in the array $\mathbf{I}$. The array $\mathbf{c}$ specifies connections between nodes; $\mathrm{c}_{i,j}$ is the node number for the `j`th input for node `i`. Inputs to a given node may originate at the current or past time steps, according to delays contained in the array $\mathbf{d}$, and through weights for time step t contained in the array $\mathbf{W}(t)$. Summarizing, the `j`th input to node `i` at time step t is the output of node $\mathrm{c}_{i,j}$ at time $t - d_{i,j}$ and is connected through weight $W_{i,j}(t)$.

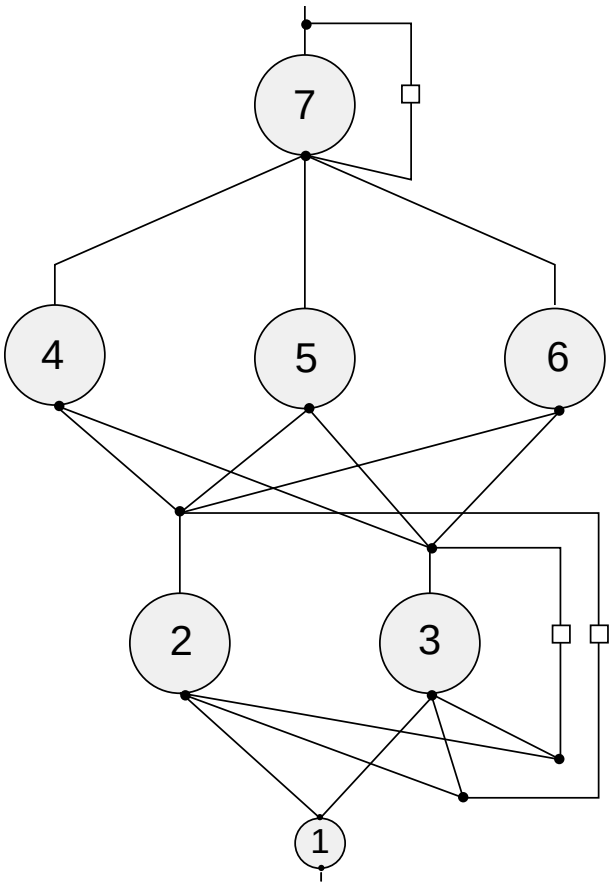
Prior to beginning network operation, all appropriate memory and internal states are initialized to zero. At the beginning of each time step, we execute the following buffer operations on weights and node outputs (in practical implementation, a circular buffer and pointer arithmetic may be employed). In the pseudocode below, `dmax` is the largest value of delay represented in the array $\mathbf{d}$ and `h` is the truncation depth of the backpropagation through time gradient calculation that is described in Section 1.3.1.

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

$$\mathbf{W}(i_t) = \mathbf{W}(i_t + 1) \tag{1.1}$$

$$y_i(i_t) = y_i(i_t + 1) \tag{1.2}$$

```
 } /* end i_t loop */
```



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

Figure 1.1: Schematic illustration of an RMLP, denoted as 1-2R-3-1R. The first hidden layer of two nodes is fully recurrent (2R in the notation), the second hidden layer (three nodes) is feedforward, and the output node is recurrent (1R in the notation). The small boxes denote unit time delays. The table contains elements of the connection and delay arrays. Entries for index i are absent for the bias (treated as node 0) and the network input (node 1), since neither receives input from any other node. Bias connections are present, as indicated in the connection table, but are not drawn. Note that recurrent connections have unit delays. The elements of the input and output arrays are: $I_1 = 1$ and $O_1 = 7$. Here activation functions of all the network nodes are assumed to be the same bipolar sigmoids. In general, we may have another table to specify the activation functions.

```
} /* end i loop */
```

Then, the actual network execution is expressed as

```
for i = 1 to n_in {
```

$$y_{\mathrm{I_i}}(t) \quad = \quad \mathrm{in}_i(t) \tag{1.3}$$

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

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

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

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

```
}
```

This description is often called an *ordered* network since the execution order of all nodes is clearly defined. Interestingly, the description does not explicitly involve the concept of *layers*; the layered structure (if desired) is imposed implicitly by the connection pattern. Pure delays can be described directly, so that tapped delay lines on either external or recurrent inputs are conveniently represented. We note that the state of the network can be compactly denoted as $\mathbf{y}(t - \Omega)$, where Ω is a set of delays $\{0, 1, \ldots, \mathtt{dmax}\}$. For example, if $\mathtt{dmax} = 2$, then the state at time t may be specified by $\mathbf{y}(t)$, $\mathbf{y}(t-1)$, and $\mathbf{y}(t-2)$.

Let us summarize the notation introduced in this section. We mark all nonscalar quantities as bold letters.

$\mathbf{y}, y_i$	Array of network node activations and a particular element.
$\mathbf{W}, W_{i,j}$	Array of weights of the network (weights of individual connections are denoted via the subscript).
$\mathbf{c}, c_{i,j}$	Connectivity table and particular element.
$\mathbf{d}, d_{i,j}$	Array of delays and one of its element.
$\mathbf{I}, I_i$	Array of indexes of nodes assigned external inputs **in**.
$\mathbf{O}, O_j$	Array of indexes of nodes used as outputs **out**.

1.3 Elements of Training

Two crucial elements of training are discussed here. We describe methods for obtaining derivatives, and then discuss weight update approaches and our preferred approach. Derivative calculations for dynamic recurrent networks have already been reviewed in Chapter ?? and are specialized to the present context here.

1.3.1 Derivative Calculations

As mentioned above, the two main approaches to obtaining derivatives are real-time recurrent learning (RTRL) and backpropagation through time (BPTT) or its truncated version BPTT(h) [Williams and Peng, 1990].

The RTRL algorithm was proposed in [Williams and Zipser, 1989] for a fully connected recurrent layer of nodes. The name RTRL is derived from the fact that the weight updates of a recurrent network are performed concurrently with network execution. The term "forward method" is more appropriate to describe RTRL, since it better reflects the mechanics of the algorithm. Indeed, in RTRL, calculations of the derivatives of node outputs with respect to weights of the network must be carried out during the forward propagation of signals in a network.

The computational complexity of RTRL scales as the fourth power of the number of nodes in a network (worst case of a fully connected TLRNN), with the space requirements (storage of all variables) scaling as the cube of the number of nodes [Williams and Zipser, 1995]. Furthermore, RTRL for a TLRNN requires that the dynamic derivatives be computed at every time step for which that TLRNN is executed. Such coupling of forward propagation and derivative calculation is due to the fact that in RTRL both derivatives and TLRNN node outputs evolve recursively. This difficulty is independent of the weight update method employed, and it may hinder practical implementation on a serial computer processor with limited speed and resources.

We have made extensive use of forward methods of derivative calculation [Narendra and Parthasarathy, 1990, Puskorius and Feldkamp, 1994]. Some time ago, however, we replaced the forward method with a form of truncated backpropagation through time (BPTT(h)) [Williams and Peng, 1990, Werbos, 1990]. By executing BPTT(h) at *every* time step with a suitably chosen truncation depth h, we obtain derivatives that closely approximate those of the forward method with greatly reduced complexity and computational effort. In a controller training problem, the truncation depth required for sufficiently accurate derivatives depends on the nature of the plant and on the desired controller efficacy.

BPTT(h) offers a number of potential advantages relative to forward methods. First, the computational complexity scales as the product of h with the square of the number of nodes. The required storage is also less than that of RTRL; it is proportional to the product of the number of nodes and the truncation depth h. Second, BPTT(h) may lead to a more stable computation of

dynamic derivatives than do forward methods because it utilizes only the most recent information in a trajectory. Third, the use of BPTT(h) permits training to be carried out asynchronously with the TLRNN execution. This feature enabled us to test our approach on real automotive hardware, as applied to on-vehicle controller training described in [Puskorius et al., 1996].

We describe here the mechanics of the particular form of BPTT(h) we have most commonly employed. (Description of other forms can be found in [Williams and Zipser, 1995, Feldkamp and Prokhorov, 1998, Feldkamp et al., 1998].) We use the Werbos notation in which $\mathrm{F\_}^{q}x$ denotes an ordered derivative of some quantity q with respect to x. In this form of BPTT, $\mathrm{F\_}^{p}x$ denotes the ordered derivative of the output of network node p with respect to x. (The reader is cautioned not to confuse the form of BPTT(h) described in this chapter with other forms [Williams and Zipser, 1995]. In particular, the form of BPTT often referred to by Werbos [Werbos, 1990] is what may be denoted as BPTT(∞). It is recovered when BPTT is performed only at the very end of a long trajectory.)

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

```
for p = 1 to n_out {
 for i = 1 to n_nodes {
  for k = 1 to n_con(i) {
```

$$\mathrm{F\_}^{p}W_{i,k} = 0 \tag{1.6}$$

```
  } /* end k loop */
  for i_t = t to t-h-1 {
```

$$\mathrm{F\_}^{p}y_i(i_t) = 0 \tag{1.7}$$

```
  } /* end i_t loop */
 } /* end i loop */
```

$$\mathrm{F\_}^{p}y_{\mathrm{O_p}}(t) = \mathrm{F}^{p}_{init} \tag{1.8}$$

$$\xi_p(t) = \mathrm{tgt}_p(t+\tau_p) - \mathrm{out}_p(t+\tau_p) \tag{1.9}$$

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

$$i_1 = t - i_h \tag{1.10}$$

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

$$j = c_{i,k} \tag{1.11}$$

$$i_2 = i_1 - d_{i,k} \tag{1.12}$$

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

$$\mathrm{F}\_^{p}W_{i,k} \mathrel{+}= y_j(i_2)\mathrm{F}\_^{p}y_i(i_1)f'(y_i(i_1)) \tag{1.14}$$

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

Here (1.6), (1.7), and (1.8) serve to initialize the derivative arrays. In training a single network, we have $\mathrm{F}^{p}_{init} = 1$ in (1.8), expressing the fact that $\frac{\partial y_{O_p}(t)}{\partial y_{O_p}(t)} = 1$. The error $\xi_p(t)$ computed in (1.9) is used in the weight update described in the next section, where the desired value of network output $\mathrm{out}_p(t+\tau_p) = y_{O_p}(t)$ is denoted as $\mathrm{tgt}_p(t+\tau_p)$. The actual backpropagation occurs in expressions (1.13) and (1.14), which derive directly from the forward propagation expression (1.4). The derivative of the activation function $f(x)$ with respect to its argument x is denoted as $f'(x)$. We have included a discount factor γ in expression (1.13), though it is often set merely to its nominal value of unity.

1.3.2 Weight Update Methods

Calculation of the derivatives $\mathrm{F}\_^{q}\mathbf{w}$ is one of the elements necessary for training a TLRNN. Another element is a weight update method. We can broadly classify weight update methods according to the amount of information used to perform an update. We write the general update equation as

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{d}(t), \tag{1.15}$$

where the direction vector $\mathbf{d}(t)$ is determined according to some rules. We obtain the well known gradient descent if $\mathbf{d}(t) = -\eta(t)\mathrm{F}\_^{q}\mathbf{w}(t)$. Here q is a chosen cost function (frequently, a sum of squared errors), and $\eta(t)$ is the (positive-valued) learning rate, which can generally be time-varying. Naturally, gradient descent is the simplest among all first-order methods of minimization for differentiable functions, and is the easiest to implement. However, it uses the smallest amount of information for performing weight updates. An imaginary plot of total error versus weight values, known as the error surface, is highly nonlinear in a typical neural network training problem, and the total error function may have many local minima. Relying only on the gradient in this case is clearly

not the most effective way to update weights. Although various modifications and heuristics have been proposed to improve the effectiveness of the first-order methods, their convergence still remains quite slow due to the intrinsically ill-conditioned nature of training problems [Haykin, 1999]. Thus, we need to utilize more information about the error surface to make the convergence of weights faster.

In differentiable minimization, the Hessian matrix, or the matrix of second-order partial derivatives of a function with respect to adjustable parameters, contains information that may be valuable for accelerated convergence. For instance, the minimum of a function quadratic in the parameters can be reached in one iteration, provided the inverse of the nonsingular positive definite Hessian matrix can be calculated. While such superfast convergence is only possible for quadratic functions, a great deal of experimental work has confirmed that much faster convergence is to be expected from weight update methods that use second-order information about error surfaces. Unfortunately, obtaining the inverse Hessian directly is practical only for small neural networks [Bishop, 1995]. Furthermore, even if we can compute the inverse Hessian, it is frequently ill-conditioned and not positive definite, making it inappropriate for efficient minimization. For TLRNN's, we have to rely on methods which build a positive definite estimate of the inverse Hessian without requiring its explicit knowledge. Such methods for weight updates belong to a family of second-order methods. For a detailed overview of the second-order methods, the reader is referred to [Haykin, 1999]. If $\mathbf{d}(t)$ in (1.15) is a product of a specially created and maintained positive definite matrix, sometimes called the approximate inverse Hessian, and the vector $-\eta(t)\mathrm{F}\underline{}^{q}\mathbf{w}(t)$, we obtain the quasi-Newton method. Unlike first-order methods which can operate in either pattern-by-pattern or batch mode, most second-order methods employ batch mode updates. In pattern-by-pattern mode, we update weights based on a gradient $\mathrm{F}\underline{}^{q}\mathbf{w}(t)$ obtained for every instance in the training set, hence the term *instantaneous gradient*. In batch mode, the index t is no longer applicable to individual instances, and it becomes associated with a training epoch. Thus, the gradient $\mathrm{F}\underline{}^{q}\mathbf{w}(t)$ is usually a sum of instantaneous gradients obtained for all training instances during the epoch t, hence the name *batch gradient*. The approximate inverse Hessian is recursively updated at the end of every epoch, and it is a function of the batch gradient and its history. Next, the best learning rate $\eta(t)$ is determined via a one-dimensional minimization procedure, called line search, which scales the vector $\mathbf{d}(t)$ depending on its influence on the total error. The overall scheme is then repeated until the convergence of weights is achieved.

Relative to first-order methods, most effective second-order methods utilize more information about the error surface at the expense of many additional calculations for each training epoch. This often renders the overall training time to be comparable to that of a first-order method. Moreover, the batch mode of operation results in a strong tendency to move strictly downhill on the error surface. As a result, weight update methods that use batch mode have limited error surface exploration capabilities and frequently tend to become trapped

in poor local minima. This problem may be particularly acute when training TRLNN's on large and redundant training sets containing a variety of temporal patterns. In such a case, a weight update method that operates in pattern-by-pattern mode would be better, since it makes the search in the weight space *stochastic*. In other words, the training error can jump up and down, escaping from poor local minima. Of course, we are aware that no batch or sequential method, whether simple or sophisticated, provides a complete answer to the problem of multiple local minima. A reasonably small value of root-mean-squared (RMS) error achieved on an independent testing set, not significantly larger than the RMS error obtained at the end of training, is a strong indication of success. Well known techniques, such as repeating a training exercise many times starting with different initial weights, are often useful to increase our confidence about solution quality and reproducibility.

Achieving an acceptable solution while preserving accelerated convergence of second-order methods requires a compromise weight update method. Such a method is the extended Kalman filter (EKF) algorithm, discussed next.

The Kalman Recursion

We have made extensive use of training that employs weight updates based on the extended Kalman filter method first proposed by [Singhal and Wu, 1989]. Unlike the aforementioned weight update methods that originate from the differentiable function optimization framework, this method treats supervised learning of a TLRNN as an optimal filtering problem. (For background material on the Kalman filter, see [Anderson and Moore, 1979, Haykin, 1996].) Its solution recursively utilizes information contained in the training data from the very beginning of the training process operating in the pattern-by-pattern mode. Weights of the TLRNN are interpreted as *states* of a dynamical system [Singhal and Wu, 1989] (not to be confused with state vector $\mathbf{y}$ of the TLRNN). This system consists of the state evolution equation in the form of a one-step time delay operator and the observer in the form of a TLRNN. The weights must be estimated, and the improved estimates become weights of the TLRNN for the next application of the Kalman recursion.

In much of our work, we have made use of a decoupled version of the EKF method [Puskorius and Feldkamp, 1994, Puskorius and Feldkamp, 1991], which we denote as DEKF. Decoupling was crucial for early practical application of the method, when speed and memory capabilities of workstations and personal computers were severely limited. At the present time, many problems are small enough to be handled without decoupling, i.e., with *global* EKF, or GEKF. In many cases, full coupling brings benefits in terms of quality of solution and overall training time. The increased time required for each GEKF update, however, remains a potential disadvantage in real-time applications.

For generality, we present the decoupled Kalman recursion; GEKF is recovered in the limit of a single weight group ($g = 1$). The weights in $\mathbf{W}$ are organized into g mutually exclusive weight groups; a convenient and effective choice, termed node decoupling, has been to group together those weights that

feed each node. Whatever the chosen grouping, the weights of group i are denoted by $\mathbf{w}_i$. The corresponding derivatives $\mathrm{F\_}^{p}\mathbf{w}_i$ of network outputs with respect to weights $\mathbf{w}_i$ are placed in `n_out` columns of $\mathbf{H}_i$.

To minimize at time step t a cost function $E = \sum_t \frac{1}{2}\boldsymbol{\xi}(t)^T\mathbf{S}(t)\boldsymbol{\xi}(t)$, where $\mathbf{S}(t)$ is a nonnegative definite weighting matrix and $\boldsymbol{\xi}(t)$ is the vector of errors, the DEKF equations are as follows [Puskorius et al., 1996]:

$$\mathbf{A}^*(t) = \left[\frac{1}{\eta(t)}\mathbf{I} + \sum_{j=1}^{g}\mathbf{H}_j^*(t)^T\mathbf{P}_j(t)\mathbf{H}_j^*(t)\right]^{-1}, \tag{1.16}$$

$$\mathbf{K}_i^*(t) = \mathbf{P}_i(t)\mathbf{H}_i^*(t)\mathbf{A}^*(t)\,, \tag{1.17}$$

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \mathbf{K}_i^*(t)\boldsymbol{\xi}^*(t)\,, \tag{1.18}$$

$$\mathbf{P}_i(t+1) = \mathbf{P}_i(t) - \mathbf{K}_i^*(t)\mathbf{H}_i^*(t)^T\mathbf{P}_i(t) + \mathbf{Q}_i(t)\,. \tag{1.19}$$

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

Similar to the quasi-Newton methods, the matrix $\mathbf{P}$ in GEKF acts as an approximation to the inverse Hessian. In the limit of zero artificial process noise in GEKF, a simple relationship exists between two matrices $\mathbf{P}$ from the beginning ($\mathbf{P}_b$) and the end ($\mathbf{P}_e$) of the same training epoch (i.e., the complete pass through all training instances) and the Gauss-Newton approximation $\mathcal{H}$ of the Hessian [Prokhorov and Feldkamp, 1998]:

$$\mathcal{H} \;=\; \mathbf{P}_e^{-1} - \mathbf{P}_b^{-1}\,. \tag{1.20}$$

In practice, the EKF recursion is typically initialized by setting the approximate error covariance matrices to scaled identity matrices, with a scaling factor of 100 for nonlinear nodes and 1000 for linear nodes. At the beginning of training, we generally set the learning rate low (the actual value depends on characteristics of the problem, but $\eta = 0.1$ is a typical value), and start with a relatively large amount of process noise, e.g., $\mathbf{Q}_i(0) = 10^{-2}\mathbf{I}$. We have previously demonstrated that taking $\mathbf{Q}$ to be nonzero accelerates training, helps to avoid poor local minima during training, and tends to maintain the necessary property of nonnegative definiteness for the approximate error covariance matrices [Puskorius and Feldkamp, 1991]. As training progresses, we generally decrease

the amount of process noise to a limiting value of approximately $\mathbf{Q}_i(t) = 10^{-6}\mathbf{I}$, and increase the learning rate to a limiting value no greater than unity. The training dynamics depend on which form of BPTT is used for derivative calculations (the values stated above are largely based on our experience with the form of BPTT described in Section 1.3.1). In addition, we have also found that occasional reinitializations of the error covariance matrices, along with resetting of initial values for the learning rate and process noise terms, may benefit the training process. Finally, one should note that initial choices for the learning rate and error covariance matrices are not independent: a multiplicative increase in the scaling factor for the approximate error covariance matrices can be canceled by reducing the initial learning rate by the inverse of the scaling factor (the relative scalings of the learning rate and error covariance matrices affect the choice of the artificial process noise term as well). The reader is referred to [Puskorius and Feldkamp, 1999] for a more detailed discussion of these issues.

Concluding our discussion of elements of training, we wish to summarize the notation introduced in this section.

$\mathrm{F}\_^{q}\mathbf{x}$	Array of ordered derivatives of quantity q with respect to $\mathbf{x}$.
$\mathrm{F}\_^{p}W_{i,k}$	Ordered derivative of the output of node p with respect to $W_{i,k}$.
ξ_p	Instantaneous error for node p between the desired output $\mathrm{tgt_p}$ and its actual value $\mathrm{out_p}$.
$\mathbf{w}$	Vector of network weights.
η	Learning rate.
$\mathbf{P}$	Approximate error covariance matrix.
$\mathbf{H}$	Matrix of derivatives of network outputs with respect to network weights.
$\mathbf{Q}$	Covariance matrix for process noise.

1.4 Basic Approach to Controller Synthesis

Our current approach is based on using a set of modular functions. We first develop a plant model using either the generating equations directly (if available; see Example 1) or a differentiable identification (ID) network trained to predict plant outputs for a number of time steps. Then we combine the controller network with the ID network and plant to form a closed-loop system. In this configuration, the correct calculation of total derivatives with respect to controller weights is somewhat intricate. Once the derivatives are obtained, weights may be updated using EKF.

1.4.1 Specifics of Controller Training

We describe controller training in the simulation setting, so that the plant is not a physical system but rather is given to us as a set of difference equations. We do not restrict our consideration to any particular type of difference equations. We are interested in addressing a fairly general class of control problems, which makes it impossible to use constructive methods of canceling or subtracting plant nonlinearities. We work with multi-input/multi-output nonlinear plants whose states are coupled with each other and with the imposed control signals. The states may also have different delays for different controls. Additional complications may include various unmeasured output and parametric disturbances, a possibly non-unique plant inverse, constraints on input and outputs, and plant states that are not directly accessible for measurement. Although we wish to keep our simulations realistic, with complexity representative of real-world problems, we can exploit abilities not possible with physical systems, such as virtually unlimited execution, arbitrary instantiation of plant states, or modification of plant parameters.

We follow the framework for neurocontrol established in [Narendra and Parthasarathy, 1990, Narendra and Parthasarathy, 1991], and we adopt the viewpoint of model-reference control, with the controller being a TLRNN. An example of the general control structure for model-reference control is shown in Figure 1.2. The plant is connected to the controller whose goal is to make the plant outputs track the desired (or reference) signals. The plant evolves as a function of the input vector $\mathbf{y}_{\mathrm{I_{pt}}}(t)$ and its internal state $\mathbf{y}_{pt}(t-\Omega_{pt})$ (not shown in the figure), where Ω_{pt} is the set of delays, introduced in Section 1.2, reflecting all delays present in the plant. The input vector $\mathbf{y}_{\mathrm{I_{pt}}}(t)$ consists of control signals $\mathbf{y}_{\mathrm{O_{cr}}}(t)$ (outputs of the controller) and any other external variables including unmeasured disturbances. The plant output may also be corrupted by measurement noise.

The controller receives the time-delayed output of the plant along with the reference signals $\mathbf{y}_{\mathrm{I_{rm}}}(t)$, which are also inputs to a reference model (see below). The feedback loop is closed through a time delay operator z^{-d}. The appropriate delays are applied component-wise to elements of the vector $\mathbf{y}_{\mathrm{O_{pt}}}(t)$. The controller produces the vector $\mathbf{y}_{\mathrm{O_{cr}}}(t)$ as a function of the input vector $\mathbf{y}_{\mathrm{I_{cr}}}(t)$, its internal state $\mathbf{y}_{cr}(t-\Omega_{cr})$, and the vector of controller weights $\mathbf{w}_{cr}$.

The desired output $\mathbf{y}_{\mathrm{O_{rm}}}(t)$ of the plant is given by the output of a stable reference model which is specified as a function of the reference signals $\mathbf{y}_{\mathrm{I_{rm}}}(t)$ and the internal state $\mathbf{y}_{rm}(t-\Omega_{rm})$ of the reference model. The goal is to train the controller network so that the errors $\boldsymbol{\xi}(t) = \mathbf{y}_{\mathrm{O_{rm}}}(t) - \mathbf{y}_{\mathrm{O_{pt}}}(t)$ are minimized over time. The simplest possible reference model is just the time delay operator, z^{-d}, providing appropriate time shifts of components of the vector $\mathbf{y}_{\mathrm{I_{rm}}}(t)$ in order to correctly compute $\boldsymbol{\xi}(t)$. Such shifts are necessary to reflect causality and properly account for internal delays always present in the plant.

We notice that the architecture of Figure 1.2 is recurrent even if the controller network is feedforward (i.e., has no internal state) due to the feedback loop. Indeed, a subset of the controller network inputs is an implicit function of controller outputs from previous time steps. Apart from the plant equations, we

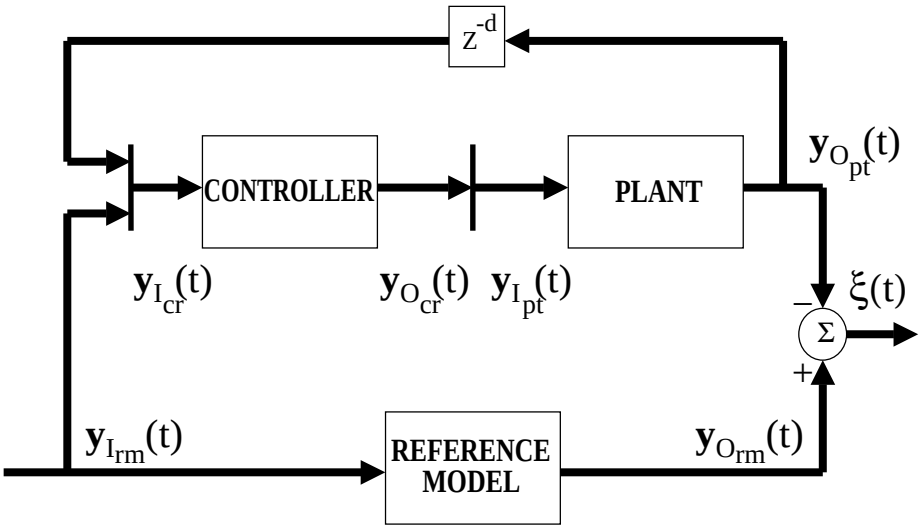


Figure 1.2: Block diagram of model reference control. The notation $\mathbf{y}_{\mathrm{O}_*}$,$\mathbf{y}_{\mathrm{I}_*}$, and $\mathbf{y}_*$ is used to distinguish between different elements of the diagram (outputs, inputs and state variables, respectively). The symbol $*$ stands for plant, controller, identification network, or reference model. Implicit in this diagram are various disturbances and noise always present in real control systems.

can interpret the entire closed-loop system as a heterogeneous TLRNN. Controller network training thus amounts to training a *subset* of weights of that TLRNN. This viewpoint neatly bypasses the following conceptual difficulty: while the plant's desired behavior is known from the outputs of the reference model, the controls leading to that behavior are not known *a priori*, but must be inferred indirectly.

In order to compute controller weight updates with dynamic gradient methods, we must estimate the total (ordered) derivatives of plant outputs, and often those of controller outputs, with respect to the weights of the controller network. As mentioned in Section 1.3.1, we use truncated backpropagation through time BPTT(h) for this purpose. Sensitivities of plant outputs to changes in the weights of the controller network, however, are not available unless another component is added to the closed-loop system of Figure 1.2. This new component is called the plant model, which provides estimates of the differential relationship of plant outputs with respect to plant inputs, previous plant outputs, and prior internal states of the plant. The plant model usually runs concurrently with the plant, receiving the vector of control signals $\mathbf{y}_{\mathrm{O}_{\mathrm{cr}}}(t)$ produced by the controller and the time-delayed outputs of the plant. The plant model outputs are then estimates of the corresponding plant outputs. The resulting closed-loop control system can be regarded as an example of indirect model-reference control [Narendra and Parthasarathy, 1990].

Of course, a suitable plant model must be created before it may be used.

Two alternatives exist for a plant model when the controller is being developed in simulation. One alternative is to use a copy of the plant's (generating) equations as the plant model. To exercise this option, however, one must be certain that the generating equations are differentiable so that BPTT(h) can be applied. Another alternative is to train a neural network to identify the relationship between plant inputs and outputs; we call the resulting model an identification (ID) network. We are more concerned with accurate estimation of the differential relationship between the plant inputs and outputs (sensitivities) than with proximity of the absolute values of the model's outputs to their respective targets. We have found that it is usually not required that the ID network be an accurate model of the plant; rather, it is important that the relevant derivatives and their temporal aspects be reasonably accurate. Thus, we sacrifice short-term plant output prediction accuracy in favor of a more accurate temporal relationship; a TLRNN is therefore a frequent choice.

Based on our experience, it is clear that obtaining correct derivatives for controller training generally requires some depth $h > 0$ in BPTT(h). Indeed, it may be shown that BPTT(0) fails to yield the correct (optimal) result for a very simple linear quadratic regulation problem [White and Sofge, 1992] regardless of training strategy. Another simple example, given in [Prokhorov and Feldkamp, 1997], requires BPTT(2) for successful training. A reasonable lower limit on the value of h is the larger of the total number of recurrent nodes in the controller network or the maximum relative degree of the plant.

1.4.2 Modular Approach

In Section 1.2 we spelled out forward and backward equations for a TLRNN. We also mentioned that controller training is akin to updating weights of a part of a heterogeneous TLRNN describing the entire closed-loop system. While representing the whole closed-loop system as a single TLRNN is possible and may even be convenient (e.g., if all the components of the closed-loop system can be represented as neural networks), we prefer to keep the components as separate entities. In our modular approach, each component is described by a separate function. Forward propagation of signals for a structure functioning in discrete time is compactly denoted as

$$\mathbf{y}_*(t) \quad = \quad \mathrm{FP}_* \left(\mathbf{y}_*(t - \Omega_*) \right), \tag{1.21}$$

where the set Ω_* is introduced to reflect causality (see Section 1.2). As in a programming language, we term FP_* a function, (1.21) being a convenient notation for the pseudocode (1.1)-(1.5) of the TLRNN execution (see Section 1.2), and use the subscript $*$ to label components of the closed-loop system. The forward propagation functions for the plant, controller, identification network, and reference model are denoted as FP_{pt}, FP_{cr}, FP_{id}, and FP_{rm}, respectively. We do not include weights $\mathbf{w}_*$ as explicit inputs to FP_*, because it is clear from the notation which weights are to be used. The set Ω_* includes delays which are determined individually for each node of a TLRNN (as in (1.4) or (1.12)).

The pseudocode (1.7)-(1.14) of Section 1.3.1 is denoted by

$$[\mathrm{F}\_\mathbf{y}^{\mathbf{q}}{}_*(t-\Omega_*), \mathrm{F}\_\mathbf{w}^{\mathbf{q}}{}_*] \quad = \quad \mathrm{BPTT}_* \left(\mathrm{F}\_\mathbf{y}^{\mathbf{q}}{}_*(t) \right). \tag{1.22}$$

The initialization in (1.6) is performed only once for each time step, and that in (1.8) is provided by the argument to the function BPTT_*. Of course, any implementation of the function BPTT_* also requires specifying $\mathbf{w}_*$ and $\mathbf{y}_*(t-\Omega_*)$ as well as the truncation depth h; for simplicity, we regard these as implicit in the function call. Here we set $h = 0$ since the reverse time loop (for $\mathtt{i_h}$ in the pseudocode of Section 1.3.1) is handled explicitly (see below). The superscript $\mathbf{q}$ denotes the quantities for which derivatives are sought.

We introduce the notation EKF_* to represent the Kalman recursion (1.16)-(1.19) (see Section 1.3.2):

$$\mathbf{w}_*(t+1) \quad = \quad \mathrm{EKF}_* \left(\mathbf{w}_*(t), \mathrm{F}\_\mathbf{w}^{\mathbf{q}}{}_*, \boldsymbol{\xi}(t), \eta(t), \mathbf{Q}(t) \right), \tag{1.23}$$

where $\mathbf{q}$ is a vector of quantities for which explicit targets are provided. For instance, $\mathbf{q}$ may be a combined vector of plant outputs and control signals. In (1.23), the approximate error covariance matrix $\mathbf{P}$ (see (1.19)) is not specified explicitly, but it should certainly be provided to make any implementation possible.

For a complete modular description, it is useful to define one more function,

$$\mathbf{y}_*(t_1) \quad = \quad \mathrm{set}\left(\mathbf{y}_*(t_2)\right), \tag{1.24}$$

which acts as a "smart" link between different functions FP_* or BPTT_*. Time indexes t_1 and t_2 represent appropriate times. In the case of forward propagation, it allows us to connect appropriate outputs of one or more functions FP_* to appropriate inputs of the receiving FP_*. In the case of backpropagation, the function $\mathrm{set}(\cdot)$ permits the assignment of the output of one BPTT_* function to initialize another BPTT_* function.

We illustrate our modular description for a typical controller training below.

```
for t = t_init to t_final {
```

$$\begin{aligned}
\mathbf{y}_{\mathrm{I_{cr}}}(t-1) &= \mathrm{set}\left(\mathbf{y}_{\mathrm{O_{pt}}}(t-\Omega_{pt}-1), \mathbf{y}_{\mathrm{I_{rm}}}(t-1)\right) && (1.25)\\
\mathbf{y}_{\mathrm{I_{id}}}(t-1) &= \mathrm{set}\left(\mathbf{y}_{\mathrm{O_{pt}}}(t-\Omega_{pt}-1)\right) && (1.26)\\
\mathbf{y}_{cr}(t-1) &= \mathrm{FP}_{cr}\left(\mathbf{y}_{cr}(t-\Omega_{cr}-1)\right) && (1.27)\\
\mathbf{y}_{\mathrm{I_{pt}}}(t) &= \mathrm{set}\left(\mathbf{y}_{\mathrm{O_{cr}}}(t-1)\right) && (1.28)\\
\mathbf{y}_{\mathrm{I_{id}}}(t) &= \mathrm{set}\left(\mathbf{y}_{\mathrm{O_{cr}}}(t-1)\right) && (1.29)\\
\mathbf{y}_{pt}(t) &= \mathrm{FP}_{pt}\left(\mathbf{y}_{pt}(t-\Omega_{pt})\right) && (1.30)\\
\mathbf{y}_{id}(t) &= \mathrm{FP}_{id}\left(\mathbf{y}_{id}(t-\Omega_{id})\right) && (1.31)\\
\mathbf{y}_{rm}(t) &= \mathrm{FP}_{rm}\left(\mathbf{y}_{rm}(t-\Omega_{rm})\right) && (1.32)
\end{aligned}$$

```
for i = 0 to h {
```

$$\left[\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{id}(t-\Omega_{id}-i), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{id}\right] = \mathrm{BPTT}_{id}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{id}(t-i)\right) \quad (1.33)$$

$$\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{\mathrm{O_{cr}}}(t-1-i) = \mathrm{set}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{\mathrm{I_{id}}}(t-i)\right) \quad (1.34)$$

$$\left[\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{cr}(t-\Omega_{cr}-1-i), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{cr}\right] = \mathrm{BPTT}_{cr}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{cr}(t-1-i)\right) \quad (1.35)$$

$$\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{\mathrm{O_{id}}}(t-1-i) = \mathrm{set}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{\mathrm{I_{cr}}}(t-1-i)\right) \quad (1.36)$$

```
} /* end i loop */
```

$$\mathbf{w}_{cr}(t+1) = \mathrm{EKF}_{cr}\left(\mathbf{w}_{cr}(t), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{cr}, \boldsymbol{\xi}(t), \eta(t), \mathbf{Q}(t)\right) \quad (1.37)$$

```
} /* end t loop */
```

Equation (1.26) demonstrates *teacher-forcing* of the plant model (ID network) by relevant plant outputs at every time step. In the control framework of Narendra and Parthasarathy this is also called the *series-parallel model* of the plant. Alternatively, (1.26) is removed if we choose the *parallel model*. In general, the series-parallel and parallel models represent alternative ways of system simulation. While the series-parallel model must be aligned at every time step with the evolution of the plant outputs, the parallel model does not require such an alignment. Preference for one model over the other is naturally based on plant behavior and complexity.

In this chapter, we do not consider concurrent adaptation of the plant model and controller. We assume that the ID network weights $\mathbf{w}_{id}$ are fixed at all times. However, we can utilize the derivatives $\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{id}$ obtained naturally during backpropagation through the ID network (see (1.33)) if its adaptation is required.

The pseudocode above certainly does not represent a computer program ready for use. For example, though implied, calculations of the vector $\boldsymbol{\xi}(t)$ necessary for (1.37) are not illustrated. We provide only the essential arguments in function calls (1.21)-(1.23), leaving for the reader to supply any other arguments required for a specific implementation.

1.4.3 Multi-Stream Training

The multi-stream procedure [Feldkamp and Puskorius, 1994] was devised to cope with the sometimes conflicting requirements of training. Consider the standard TLRNN 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. In many training problems, especially those in which exogenous inputs are present, the data sequence is heterogeneous. For example, regions of rapid variation of inputs and outputs may be followed by regions of slow change. Or a sequence of outputs that centers about one level

may be followed by one that centers about a different level. For any of these cases, in a straightforward training process the tendency always exists for the network weights to be adapted unduly in favor of the most recently presented data (see Chapter **??**). This *recency effect* is analogous to the difficulty that may arise in training feedforward networks if the training data are repeatedly presented in the same order.

For feedforward networks, 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 sub-sequences, making an update only for the last input-output pair of the sub-sequence (when the network would be expected to be independent of its initialization at the beginning of the sequence). A full batch update involves running the network through the entire data set, computing the required derivatives that correspond to each input-output pair, and making an update based on the entire set of errors.

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

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

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

If we take $N_s = 1$ and $N_t - N_p = 1$, we recover the order-scrambling procedure described above; N_t may be identified with the sub-sequence 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 $N_t - N_p = 1$.

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

An important consideration is how to carry out the EKF update procedure. If first-order gradient updates were being used, we would simply average the updates that would have been performed had the streams been treated separately. In the case of EKF training, however, averaging separate updates is incorrect. Instead, we treat this problem as that of training a single shared-weight network with $N_s \times$ `n_out` outputs. From the standpoint of the EKF method, we are simply training a multiple output network in which the number of original outputs is multiplied by the number of streams. The nature of the Kalman recursion is then to produce weight updates which are not a simple average of updates that would be computed separately for each output, as is the case for a simple gradient descent weight update.

In single-stream EKF training (see Section 1.3.2), we place derivatives of network outputs with respect to network weights in the matrix $\mathbf{H}$ constructed from `n_out` column vectors, each of dimension equal to the number of trainable weights, N_w. In multi-stream training, the number of columns is correspondingly increased to $N_s \times$ `n_out`. Similarly, the vector of errors $\boldsymbol{\xi}$ has $N_s \times$ `n_out` elements. Apart from these augmentations of $\mathbf{H}$ and $\boldsymbol{\xi}$, the form of the Kalman recursion is unchanged.

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

The multi-stream procedure clearly has many practical uses for controller training. For example, multi-stream EKF can be used in conjunction with BPTT(h) as a means to avoid the recency phenomenon. In this scenario, two or more trajectories of the plant outputs and associated control signals (as generated by a single controller for a given plant) for different regions of operation are processed by the multi-stream procedure to generate one change of weights. This procedure is then iterated. This scheme is most conveniently employed when the plant can easily be driven (e.g., by a reference input) to various regions of operations.

Multi-stream EKF controller training provides the capability to train a single neurocontroller for best compromise over a range of systems, e.g., to handle plant-to-plant variability. The procedure is used to generate a robust controller by simultaneously accessing multiple instances of a plant. A set of identical controller networks (which may also be viewed as a single shared-weight network) is used to control several plants. The plants are somehow chosen to span the expected range of plant parameter variations. In the training process, several trajectories, one from each system (comprised of the controller, an ID network, and a corresponding plant), are gathered, separate dynamic gradients are computed for each of the streams, and these gradients are then processed simultaneously with the EKF weight update method to generate a new weight vector $\mathbf{w}_{cr}$ for the single controller network employed for all plant variations. This procedure is iterated until an effective compromise controller emerges. We have demonstrated in simulation the use of multi-stream DEKF for training robust engine idle speed neurocontrollers in the fashion just described [Feldkamp and Puskorius, 1994]. Another illustration of the utility of multi-streaming is given in Section 1.5. Concluding this section, we note that training controllers with multi-streaming implies the same modifications to the standard Kalman recursion (1.16)-(1.19) as those mentioned above.

1.5 Example 1

This section consists of four parts. Section 1.5.1 introduces an example control synthesis problem. Sections 1.5.2 and 1.5.3 discuss this problem for two cases which differ in the amount of information assumed to be available. Section 1.5.4 describes some improvements to the controller's performance and summarizes the results.

1.5.1 MIMO Control Problem

The plant for this example is a third-order system with two inputs and two outputs described by the state equations

$$\begin{aligned} y_{pt1}(t) &= \alpha_1 y_{pt1}(t-1)\sin\left(\alpha_2 y_{pt2}(t-1)\right) \\ &\quad + \left(\alpha_3 + \alpha_4 \frac{y_{pt1}(t-1)y_{pt4}(t)}{1+y_{pt1}^2(t-1)y_{pt4}^2(t)}\right) y_{pt4}(t) \\ &\quad + \left(\alpha_5 y_{pt1}(t-1) + \frac{\alpha_6 y_{pt1}(t-1)}{1+y_{pt1}^2(t-1)}\right) y_{pt5}(t)\,, \qquad (1.38) \\ y_{pt2}(t) &= y_{pt3}(t-1)\left(\alpha_7 + \alpha_8 \sin\left(\alpha_9 y_{pt3}(t-1)\right)\right) \\ &\quad + \frac{\alpha_{10} y_{pt3}(t-1)}{1+y_{pt3}^2(t-1)}\,, \qquad (1.39) \\ y_{pt3}(t) &= \left(\alpha_{11} + \alpha_{12}\sin\left(\alpha_{13} y_{pt1}(t-1)\right)\right) y_{pt5}(t)\,, \qquad (1.40) \end{aligned}$$

Param.	Value	Param.	Value	Param.	Value	Param.	Value
α_1	0.9	α_2	1.0	α_3	2.0	α_4	1.5
α_5	1.0	α_6	2.0	α_7	1.0	α_8	1.0
α_9	4.0	α_{10}	1.0	α_{11}	3.0	α_{12}	1.0
α_{13}	2.0						

Table 1.1: Nominal values of parameters $\boldsymbol{\alpha}$ for the MIMO plant of [Narendra and Mukhopadhyay, 1994].

where y_{pt1}, y_{pt2} and y_{pt3} are components of the plant state vector, and the plant inputs y_{pt4} and y_{pt5} are set equal to the control signals. The first two components of the state vector, y_{pt1} and y_{pt2}, are taken to be the plant outputs. The goal is to develop a neurocontroller such that the plant outputs follow two independent reference model outputs, $y_{\mathrm{O_{rm1}}}$ and $y_{\mathrm{O_{rm2}}}$, as closely as possible.

This multiple-input/multiple-output (MIMO) control problem was originally proposed by [Narendra and Mukhopadhyay, 1994]. They studied five cases of the problem formulation with constant values of the parameters $\boldsymbol{\alpha}$. These cases differ by the amount of information about the plant available to the control designer. For example, in one case all state variables were accessible and the plant equations were known. In the most complicated case, a neurocontroller was developed using only input-output data, the plant equations treated as unknown.

Table 1 contains nominal values of the plant parameters $\boldsymbol{\alpha}$. We modify the problem by introducing uncertainty in the values of $\boldsymbol{\alpha}$. More specifically, we allow each components of $\boldsymbol{\alpha}$ to be a random variable uniformly distributed around its nominal value in the range $\pm 20\%$. We then consider the two cases of the original problem formulation mentioned above, i.e., the case in which all state variables are accessible and plant equations are known, and the case in which only input-output data is available. In both cases, our controller synthesis approach can be termed *training for robustness* [Feldkamp and Puskorius, 1994]. First, we specify or train a plant model. Next, we initiate multi-stream controller training using a particular reference model and a training strategy. We test the resulting controller on many plants with different parameters $\boldsymbol{\alpha}$ and various reference signals.

Training is performed with a reference model driven by a *skyline* training pattern [Puskorius and Feldkamp, 1994] (a piece-wise constant reference trajectory), generated by

```
{ /* Reference model driven by skyline reference signals */
if (k1 = 0) { k1 = rand(10,50), r1 = rand(-1.5,+1.5) }
if (k2 = 0) { k2 = rand(10,50), r2 = rand(-1.5,+1.5) }
```
$y_{\mathrm{I_{rm1}}}(t)$ = r_1
$y_{\mathrm{I_{rm2}}}(t)$ = r_2
$y_{\mathrm{O_{rm1}}}(t)$ = $y_{\mathrm{I_{rm1}}}(t-1)$

$y_{\mathrm{O}_{\mathrm{rm}2}}(t)$ = $y_{\mathrm{I}_{\mathrm{rm}2}}(t-2)$
k_1 = k_1 - 1
k_2 = k_2 - 1
}

where rand(l,u) is a function call to a uniform random number generator returning a value between l and u (integer for rand(10,50) and real for rand(-1.5,+1.5)). The vectors $\mathbf{r} = [r_1, r_2]^T$ and $\mathbf{k} = [k_1, k_2]^T$ should be stored, and the skyline equations require k_1 and k_2 to be initialized to zero. The variables $y_{\mathrm{I}_{\mathrm{rm}1}}(t)$ and $y_{\mathrm{I}_{\mathrm{rm}2}}(t)$ are used as controller inputs, and the variables $y_{\mathrm{O}_{\mathrm{rm}1}}(t)$ and $y_{\mathrm{O}_{\mathrm{rm}2}}(t)$ are used to compute the error vector $\boldsymbol{\xi}(t)$. Different delays between each control signal and each plant output are reflected in the reference model, taking into account causality and relative degrees for each output [Narendra and Mukhopadhyay, 1994]. If the first and second components of $\boldsymbol{\xi}(t)$ correspond to outputs $y_{pt1}(t)$ and $y_{pt2}(t)$, respectively, then $\xi_1(t) = y_{\mathrm{O}_{\mathrm{rm}1}}(t) - y_{pt1}(t)$ and $\xi_2(t) = y_{\mathrm{O}_{\mathrm{rm}2}}(t) - y_{pt2}(t)$.

1.5.2 State Variables Accessible and Plant Equations Known

We assume here that the controller is an RMLP. To enable controller training and obtain total derivatives of the plant outputs with respect to controller weights, we employ here a copy of the plant equations. We choose five copies, where each copy has the same values of $\boldsymbol{\alpha}$ as the corresponding plant. We arrange training into five streams ($N_s = 5$), one stream for each plant-model-controller system. Each stream consists of a length N_t of 200 time steps with its own unique reference trajectory formed by the skyline reference signals (see Section 1.4.3 for notation). The priming length N_p is 10 time steps. In the description of the training strategy below, we define one epoch as consisting of $N_s \times (N_t - N_p) = 950$ weight updates. At the beginning of each epoch, we generate a different pair of skyline reference signals for each stream. Our typical training strategy for BPTT(9) and GEKF is to train for 300 epochs with $\eta = 0.001$ and $\mathbf{Q} = 10^{-2}\mathbf{I}$, then 300 epochs with $\eta = 0.01$ and $\mathbf{Q} = 10^{-3}\mathbf{I}$, and 300 more epochs with $\eta = 0.1$ and $\mathbf{Q} = 10^{-4}\mathbf{I}$. We conclude training with 500 additional epochs with $\eta = 1$ and $\mathbf{Q} = 10^{-5}\mathbf{I}$ (1400 epochs total). Training is carried out with the skylines (see Section 1.5.1). During the first 10 epochs of training, a fixed trajectory of skyline reference signals is used for each stream. This is done to reduce the chance of instability when controller weights have not yet been subjected to enough training. We begin training on five different plants whose parameters are drawn from the uniform distribution specified above. We keep the same five plants for 20 epochs, then obtain a new set of five plants drawn from the same distribution and continue training on them, and so on. Not changing plant parameters for some number of epochs is a reasonable compromise between the amount of learning performed on a particular plant with a variety of skylines and the total number of plants presented during training. 350 distinct plants are used during the course of training. Once training is completed, we test the resulting controller on 10,000 additional plants using a

different set of reference signals [Narendra and Mukhopadhyay, 1994]:

{ /* Reference signals for testing */
r_1 = 0.75 sin$\left(\frac{2\pi t}{50}\right)$ + 0.75 sin$\left(\frac{2\pi t}{10}\right)$
r_2 = 0.75 sin$\left(\frac{2\pi t}{30}\right)$ + 0.75 sin$\left(\frac{2\pi t}{20}\right)$
}

Our experiments indicate that the performance of recurrent controllers is substantially superior to that of feedforward controllers. Figure 1.3 (left panel) shows a histogram of the RMS test error distribution for the same 10,000 plants for the 5-20-10-2L feedforward controller (shorthand notation corresponding to a network with 5 inputs, 20 nodes in the first hidden layer, 10 nodes in the second hidden layer, and 2 linear outputs) and for the 5-10R-10-2L recurrent controller, trained using the same strategy. The statistics of these two RMS error distributions are: mean errors $\mu = 0.247$ and 0.179, standard deviations $\sigma = 0.0524$ and 0.0374, maximum RMS errors $\text{RMSE}_{\max} = 0.558$ and 0.379, and minimum RMS errors $\text{RMSE}_{\min} = 0.134$ and 0.118; the first and second values correspond to the feedforward and recurrent controllers, respectively. The two distributions are well separated, and the distribution for the recurrent controller is clearly preferable.

1.5.3 State Variables Inaccessible and Plant Equations Unknown

In the case of control using input-output data when plant equations are unknown, we need to develop a sufficiently good ID network. We begin by gathering input-output pairs for training an ID network. For the plant with nominal parameters $\boldsymbol{\alpha}$, we generate a trajectory composed of 5000 pairs $(\mathbf{in}, \mathbf{out})$, where $\mathbf{in} = [y_{pt1}(t-1), y_{pt2}(t-1), \mathbf{y}_{\text{O}_{\text{cr}}}(t)]$, $\mathbf{out} = [y_{pt1}(t), y_{pt2}(t)]$. The control signals $\mathbf{y}_{\text{O}_{\text{cr}}}(t)$ are generated as the skyline reference signals above, except that their durations are much shorter (rand(1,5)) and the maximum amplitudes are 0.8 and 0.3 for the first and second control signals, respectively. In spite of the relatively narrow range of control signal changes, the outputs $y_{pt1}(t)$ and $y_{pt2}(t)$ exhibit rapid fluctuations in the ranges (-16.0, 20.0) and (-3.0, 1.5), respectively.

We train a 4-10R-10-2L network using node decoupled EKF with five streams of 1000 points each, using the following strategy. We first train for 100 epochs with $\eta = 0.01$ and $\mathbf{Q} = 10^{-2}\mathbf{I}$. The next 100 epochs utilize $\eta = 0.1$ and $\mathbf{Q} = 10^{-3}\mathbf{I}$. We complete training with 100 more epochs using $\eta = 1$ and $\mathbf{Q} = 10^{-4}\mathbf{I}$. During training, the RMS error dropped to 1/7 of its original level. To determine when to stop the training process, we used the standard technique of measuring performance for an independent validation set.

We now provide a detailed modular description for controller synthesis. It is illustrated for the case when both the controller and the ID network are the same architecture, 4-10R-10-2L. The input nodes of these RMLP's are indexed 1 through 4, whereas the output nodes have indexes 25 and 26. The vector $\mathbf{q}$

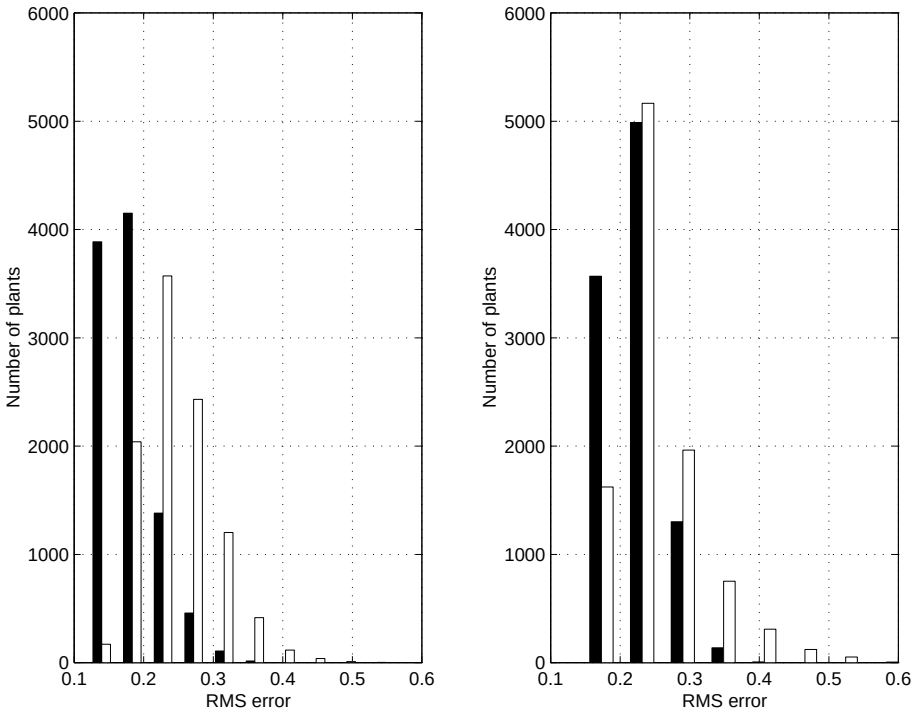


Figure 1.3: Distribution of the RMS error values for the feedforward controller (white) and the recurrent controller (black) during testing on 10,000 plants. The left panel shows the RMS error distribution for controllers synthesized in the full state feedback case (Section 1.5.2). The right panel illustrates the RMS error distribution for controllers trained with the ID network (Section 1.5.3). For the feedforward controller trained with the ID network, six plants (0.06% of total) had their RMS errors larger than 0.6 and were left out of the histogram.

consists of y_{pt1} and y_{pt2}. The set Ω_* is $\{0, 1\}$.

```
for t = t_init to t_final {
```

$$
\begin{aligned}
y_{cr1}(t-1) &= y_{pt1}(t-1) && (1.41)\\
y_{cr2}(t-1) &= y_{pt2}(t-1) && (1.42)\\
y_{cr3}(t-1) &= y_{\mathrm{I}_{\mathrm{rm}1}}(t-1) && (1.43)\\
y_{cr4}(t-1) &= y_{\mathrm{I}_{\mathrm{rm}2}}(t-1) && (1.44)\\
y_{id1}(t-1) &= y_{pt1}(t-1) && (1.45)\\
y_{id2}(t-1) &= y_{pt2}(t-1) && (1.46)\\
\mathbf{y}_{cr}(t-1) &= \mathrm{FP}_{cr}\left(\mathbf{y}_{cr}(t-\Omega_{cr}-1)\right) && (1.47)\\
y_{id3}(t) &= y_{cr25}(t-1) && (1.48)\\
y_{id4}(t) &= y_{cr26}(t-1) && (1.49)\\
\mathbf{y}_{id}(t) &= \mathrm{FP}_{id}\left(\mathbf{y}_{id}(t-\Omega_{id})\right) && (1.50)\\
y_{pt4}(t) &= y_{cr25}(t-1) && (1.51)\\
y_{pt5}(t) &= y_{cr26}(t-1) && (1.52)\\
\mathbf{y}_{pt}(t) &= \mathrm{FP}_{pt}\left(\mathbf{y}_{pt}(t-\Omega_{pt})\right) && (1.53)\\
\mathbf{y}_{rm}(t) &= \mathrm{FP}_{rm}\left(\mathbf{y}_{rm}(t-\Omega_{rm}-1)\right) && (1.54)\\
\xi_1(t) &= y_{O_{rm1}}(t) - y_{pt1}(t) && (1.55)\\
\xi_2(t) &= y_{O_{rm1}}(t) - y_{pt2}(t) && (1.56)
\end{aligned}
$$

```
for i = 0 to h {
```

$$
\begin{aligned}
\left[\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{id}(t-\Omega_{id}-i), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{id}\right] &= \mathrm{BPTT}_{id}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{id}(t-i)\right) && (1.57)\\
\mathrm{F}^{\mathbf{q}}_{\_}y_{cr25}(t-1-i) &= \mathrm{F}^{\mathbf{q}}_{\_}y_{id3}(t-i) && (1.58)\\
\mathrm{F}^{\mathbf{q}}_{\_}y_{cr26}(t-1-i) &= \mathrm{F}^{\mathbf{q}}_{\_}y_{id4}(t-i) && (1.59)\\
\left[\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{cr}(t-\Omega_{cr}-1-i), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{cr}\right] &= \mathrm{BPTT}_{cr}\left(\mathrm{F}^{\mathbf{q}}_{\_}\mathbf{y}_{cr}(t-1-i)\right) && (1.60)\\
\mathrm{F}^{\mathbf{q}}_{\_}y_{id25}(t-1-i) &= \mathrm{F}^{\mathbf{q}}_{\_}y_{cr1}(t-1-i) && (1.61)\\
\mathrm{F}^{\mathbf{q}}_{\_}y_{id26}(t-1-i) &= \mathrm{F}^{\mathbf{q}}_{\_}y_{cr2}(t-1-i) && (1.62)
\end{aligned}
$$

```
} /* end i loop */
```

$$
\mathbf{w}_{cr}(t+1) = \mathrm{EKF}_{cr}\left(\mathbf{w}_{cr}(t), \mathrm{F}^{\mathbf{q}}_{\_}\mathbf{w}_{cr}, \boldsymbol{\xi}(t), \eta(t), \mathbf{Q}(t)\right) \quad (1.63)
$$

```
} /* end t loop */
```

In the modular description above, calls to the function set$(\cdot)$ are replaced by appropriate assignments between the nodes of the networks and the plant outputs. We also illustrate how to calculate components of the error vector $\boldsymbol{\xi}(t)$ for each stream (see (1.55) and (1.56)).

We carry out training for robustness as described in Section 1.5.2. The differences are related to the training strategy and the use of an ID network. We choose to restrict η to 0.1 for the last 500 training epochs for reasons we mention below. Regarding the use of the ID model, we point out a crucial difference between the setting of Section 1.5.2 and the current experiment. In Section 1.5.2, each copy of the plant equations used as the plant model has the same values of $\boldsymbol{\alpha}$ as those of the corresponding perturbed plant. Here we use the same ID network with *fixed* weights for the entire training session.

For comparison, we also train a 4-20-10-2L feedforward controller for robustness using the same settings and training strategy. Our preliminary experiments showed that the training process may become unstable if the learning rate is increased to $\eta = 1$. We were unable to destabilize training of the recurrent controller, but for proper comparison we decided to restrict the maximum learning rate for its training as well.

As in Section 1.5.2, our results show that recurrent controllers are substantially superior to feedforward controllers in terms of robustness to perturbations of plant parameters. Figure 1.3 (right panel) shows a histogram of the RMS test error distribution for the same 10,000 plants for the 4-20-10-2L feedforward controller and for the 4-10R-10-2L recurrent controller trained using the same strategy with BPTT(9). The statistics of the two RMS error distributions are: $\mu = 0.253$ and 0.221, $\sigma = 0.0611$ and 0.0374, $\text{RMSE}_{\max} = 0.729$ and 0.400, and $\text{RMSE}_{\min} = 0.161$ and 0.144, corresponding to the feedforward and recurrent controllers, respectively.

It is also instructive to train a recurrent controller on the nominal plant only, using the same training strategy as for the controllers trained for robustness, and then test it on many different plants. On the nominal plant, this controller has an RMS error of 0.082, compared to 0.158 for the recurrent controller trained for robustness. On the other hand, the recurrent controller trained only on the nominal plant has significantly higher probability for failure in the robustness test. On 10,000 randomly generated plants, its maximum RMS error is 0.543, growing to 0.923 in a test with one million plants, as compared to 0.443 for the recurrent controller trained for robustness. Here we observe a tradeoff between robustness to plant variations and the minimum tracking error achievable on a given plant. Similar observations were made for the feedforward controller.

Concluding this section, we would like to point out that obtaining an acceptable ID network of the nominal plant does not appear to be a challenging task. Our ID network turned out to be sufficiently accurate to enable training of controllers for robustness. Of course, it would be possible to train an ID network specialized to each plant and used in the multi-stream training framework. The computational and logistical complexity of such an experiment, however, is staggering since a separate ID network would have to be trained for every new plant drawn from the distribution. Training just a small number of ID networks for a selected set of plants may be a viable alternative. This would be similar to system identification of a plant capable of operating in several distinct modes, and these modes could be singled out for identification and control purposes. Another alternative is to train a "universal" ID network, with the

plant parameters $\boldsymbol{\alpha}$ as additional inputs.

1.5.4 Further Testing, Improvements and Summary of the Results

The controllers obtained in Sections 1.5.2 and 1.5.3 were subjected to further testing on more difficult reference signals crafted to test the degree of decoupling between the two outputs. Figures 1.4 and 1.5 illustrate the behavior of the outputs of the nominal plant controlled by the recurrent and feedforward controllers, respectively, in the case of full state feedback control (five inputs to each controller). Each output must follow its own reference signal as closely as possible. Each reference signal trajectory consists of 10 segments of 100 time steps. Only the second segment coincides with reference signals previously used for testing. The results obtained with the feedforward controller in the closed loop are substantially inferior to those obtained with the recurrent controller (RMS errors 0.369 and 0.208, respectively). Statistics of the RMS error distribution obtained for these 1000-step reference signals and 10,000 plants drawn from the same uniform distribution as before become even more saliently in favor of the recurrent controller: $\mu = 0.444$ and 0.278, $\sigma = 0.0848$ and 0.0549, $\mathrm{RMSE}_{\max} = 0.872$ and 0.571, and $\mathrm{RMSE}_{\min} = 0.249$ and 0.169, for the feedforward and recurrent controllers, respectively.

We also tested the controllers obtained in Section 1.5.3 on the 1000-step reference signals used above. Figure 1.6 shows behavior of the nominal plant controlled by the four-input recurrent controller, with an RMS error of 0.272. Spikes and high-frequency oscillations are quite apparent, and it is desirable to suppress them. Although we tested the four-input feedforward controller, we do not show these results due to poor quality (the controller nearly lost stability on the last segment; high-frequency oscillations of outputs on other segments were increased as well).

We were able to reduce the number of spikes while improving the overall robustness of the resulting recurrent controller by employing a different set of reference signals and a slightly modified training strategy. After the first 100 epochs, we switched from skylines to uniformly distributed random noise in the range ± 1.5. We encountered frequent loss of stability when using a learning rate η larger than 0.01, so we modified the training strategy as follows. The first 400 epochs we trained with $\eta = 0.001$ and $\mathbf{Q} = 10^{-2}\mathbf{I}$, then 500 epochs with $\eta = 0.01$ and $\mathbf{Q} = 10^{-3}\mathbf{I}$, 500 more epochs with $\eta = 0.01$ and $\mathbf{Q} = 10^{-4}\mathbf{I}$, and the final 600 epochs with $\eta = 0.01$ and $\mathbf{Q} = 10^{-5}\mathbf{I}$ (2000 epochs total). Thus we trained on a total of 500 distinct plants. The rest of the training strategy (the number of streams, the choice of generating a new set of plant parameters every 20 epochs, etc.) remained the same. The test performance of the resulting controller is shown in Figure 1.7, with an RMS error of 0.230.

The statistics of the RMS error distribution obtained for the four-input recurrent controller trained on random reference signals for the 1000-step test reference signals and 10,000 randomly perturbed plants are substantially better than those of the recurrent controller trained on skylines, perhaps due to the

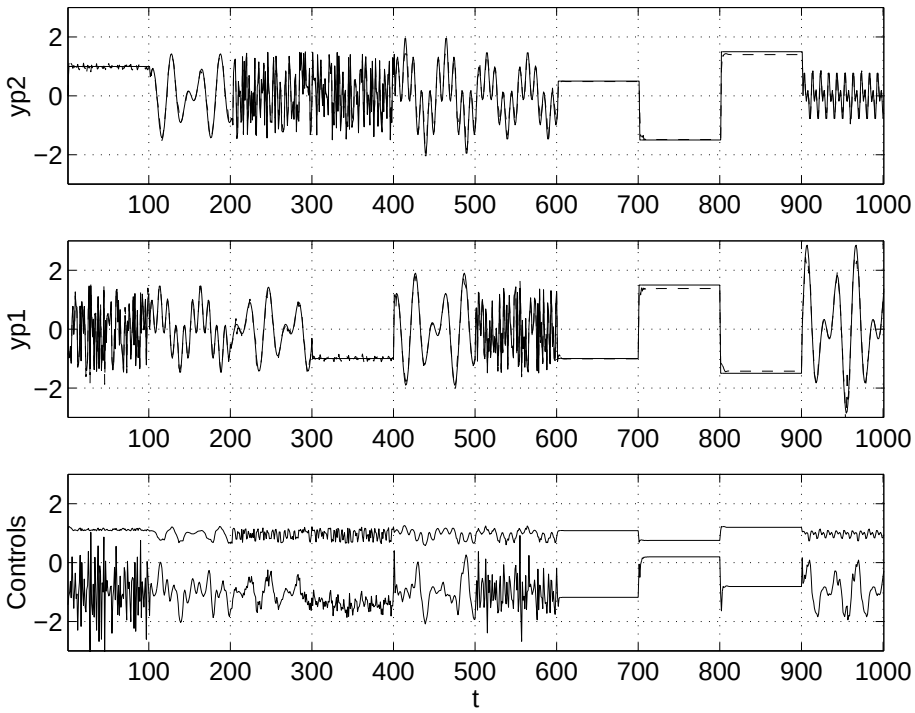


Figure 1.4: Performance of the recurrent controller during testing in the full state feedback case. The actual plant outputs, y_{pt1} (middle panel) and y_{pt2} (upper panel), are shown as dashed lines. The desired outputs, $y_{\mathrm{O_{rm1}}}$ and $y_{\mathrm{O_{rm2}}}$, are drawn as solid lines. For clarity, the two outputs of the controller (lower panel) are shifted, the lower curve (the first control) by -1.0 and the upper curve (the second control) by +1.0.

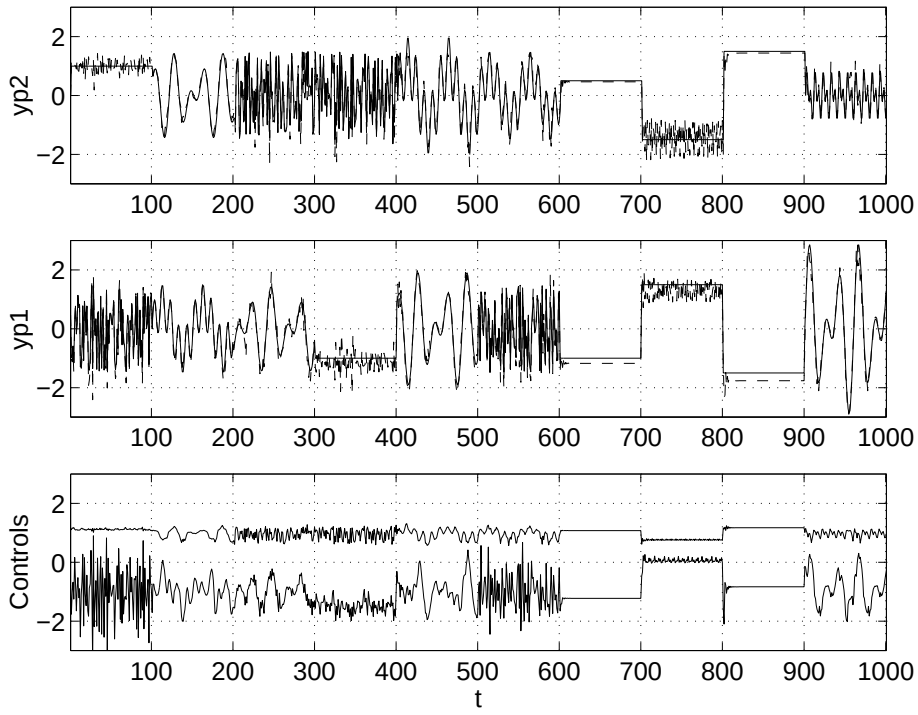


Figure 1.5: Performance of the feedforward controller during testing in the full state feedback case. The layout is the same as in the previous figure.

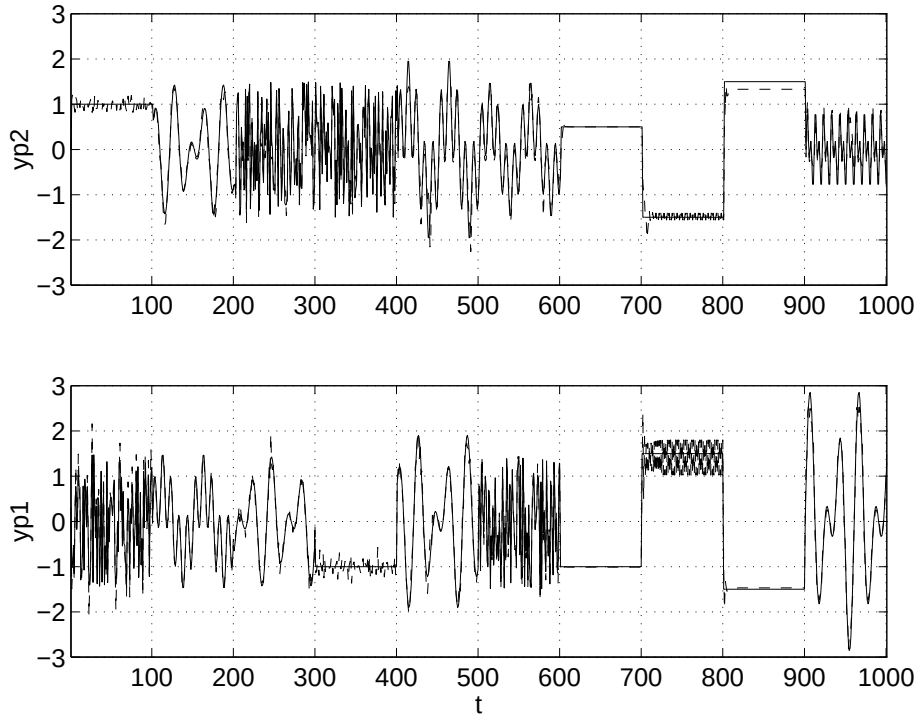


Figure 1.6: Test performance of the recurrent controller trained using the ID network. The actual plant outputs, y_{pt1} (lower panel) and y_{pt2} (upper panel), are shown as dashed lines. The desired outputs, y_{Orm_1} and y_{Orm_2}, are drawn as solid lines.

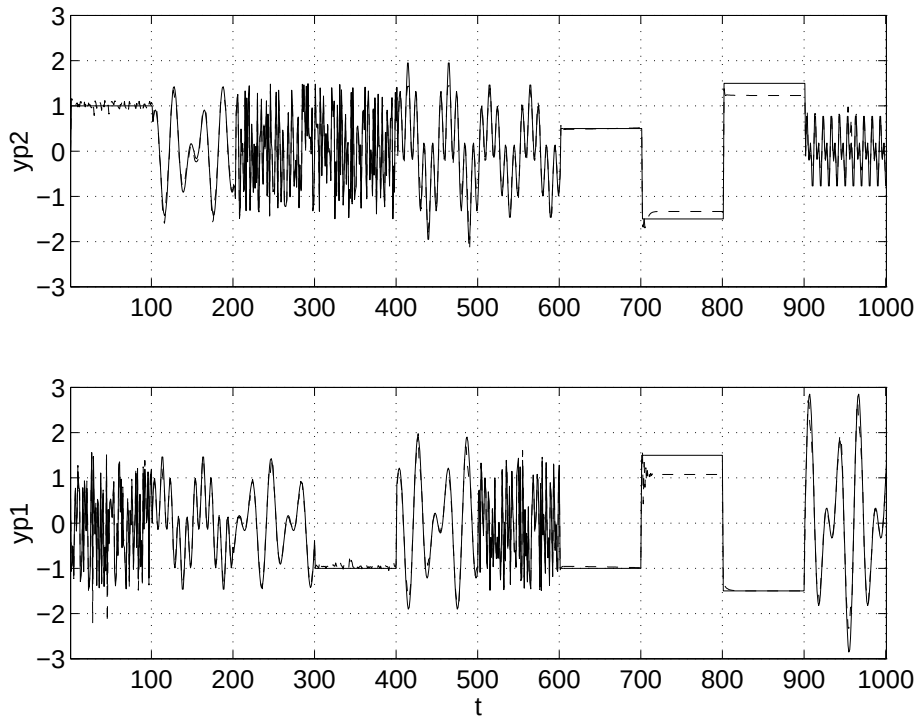


Figure 1.7: Test performance of the recurrent controller trained with random reference signals using the ID network. The actual plant outputs, y_{pt1} (lower panel) and y_{pt2} (upper panel), are shown as dashed lines. The desired outputs, $y_{\mathrm{O_{rm1}}}$ and $y_{\mathrm{O_{rm2}}}$, are drawn as solid lines.

greater amount of excitation present in completely random reference signals relative to skyline signals. These statistics are also similar to those obtained for the recurrent controller in the full state feedback case: $\mu = 0.303$ and 0.278, $\sigma = 0.0479$ and 0.0549, $\mathrm{RMSE_{max}} = 0.550$ and 0.571, and $\mathrm{RMSE_{min}} = 0.206$ and 0.169, for the recurrent controllers trained with the random reference signals and in the full state feedback case, respectively.

We have carried out many training experiments for recurrent and feedforward controllers in both cases. Our conclusions are summarized below.

It is not very challenging to train a very good controller (recurrent or feedforward) on a plant with any specific set of parameters $\boldsymbol{\alpha}$, either nominal or drawn from the range $\pm 20\%$ around nominal (see Figure 1.8 for typical test results). Such a controller, however, has unacceptable performance when tested for robustness (the mean error is usually larger and the RMS error distribution is at least two or three times wider than for a controller trained for robustness). When trained on a particular plant and tested on the same plant, a recurrent controller generally results in plant behavior with more spikes than that ob-

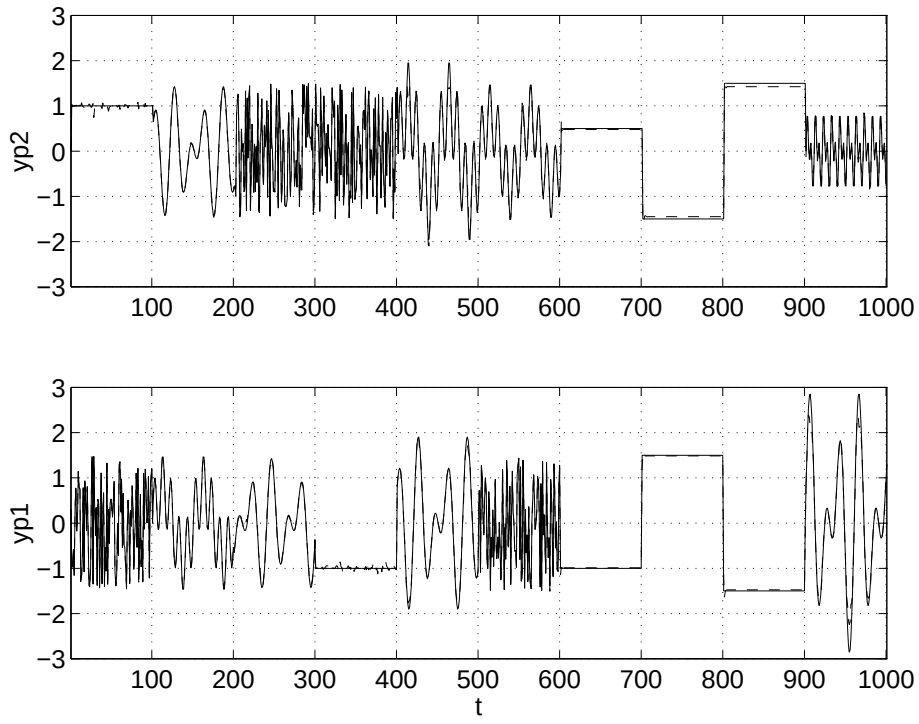


Figure 1.8: Test performance for the 5-20-10-2L feedforward controller trained with skylines on the nominal plant. The test is carried out on the same plant. The actual plant outputs, y_{pt1} (lower panel) and y_{pt2} (upper panel), are shown as dashed lines. The desired outputs, $y_{\mathrm{O}_{\mathrm{rm}1}}$ and $y_{\mathrm{O}_{\mathrm{rm}2}}$, are drawn as solid lines.

tained with a feedforward controller. This should not be very surprising, since the closed-loop system including a recurrent controller has many more degrees of freedom than a system with a feedforward controller.

We are aware of a certain degree of arbitrariness in our choice of the controller network architecture (numbers of nodes and layers) and training parameters. The latter includes elements of the training strategy (lengths of training epochs with chosen values of η and $\mathbf{Q}$, numbers of streams and their lengths) and particular ways of choosing plants and reference signals. We can partially justify our choice of the network architecture and training parameters. For example, our training strategy (increasing the learning rate η while decreasing diagonal elements of the matrix $\mathbf{Q}$ during training), along with various forms of skyline reference signals, has been reliable and worked well, not only in this example application, but also in other problems we have dealt with in the past.

Our experiments convincingly indicate that training using only one stream does not give rise to a robust controller, because of the recency effect. In this case, even longer training does not help. Utilizing more than five streams may

improve the RMS error statistics (particularly, maximum RMS errors), as does an increased size of controller network or longer truncation depth in BPTT(h). Such improvements, however, do not appear to be dramatic, and they all come at a price of a (sometimes substantial) increase in training time. We find that proper comparison of training strategies differing in the total number of plant variations presented to the controller requires prorating the total number of training epochs accordingly. For instance, 5-stream training requires twice as many epochs as does 10-stream training to achieve about the same level of performance in testing, all other training parameters being equal.

Introducing full recurrence into the second layer of the recurrent controller appears only to result in greater occurrences of spikes and high-frequency oscillations on smooth test signals. On the other hand, networks smaller than the one chosen (e.g., a controller 4-8R-5R-2L) resulted in at least 50% larger RMS errors than the best numbers presented here.

To train for robustness, we chose individual plants at random from the assumed range around the nominal values of the parameters $\boldsymbol{\alpha}$. We did not analyze the properties of individual plants with different sets of $\boldsymbol{\alpha}$ with the intention of selecting several plants for further multi-stream training (i.e., to select a "magic" combination of plants). In our training sessions, we changed the reference signals every epoch while keeping five distinct sets of $\boldsymbol{\alpha}$ constant for 20 epochs (five streams). Our intention was to provide a reasonable balance between the variety of reference signals and the variety of plants presented in the course of training. We allowed the controller to get a good handle on the plants' behavior on various reference trajectories before the next set of plant variations was generated (every 20 epochs; see Section 1.5.2). The robustness test revealed poor results when the five plants were sampled from the uniform distribution either too often (e.g., every epoch) or too seldom (e.g., every 100 epochs), while the other training parameters remained the same. In the former case, however, good results could still be obtained provided that training continued for much longer.

The results presented here confirm our previous observations about the superior robustness properties of recurrent neurocontrollers trained with EKF-based techniques [Feldkamp and Puskorius, 1994]. The technique for training for robustness described here represents a successful method for reducing the standard deviation of the RMS error distribution as well as decreasing the probability of worst case behavior. Nevertheless, it is entirely possible that there exists a recurrent neurocontroller architecture and a set of training parameters that will deliver results better than those presented here.

1.6 Example 2

In this section, we consider the problem of financial portfolio optimization from the perspective of neurocontrol. In particular, we demonstrate that the methods described earlier in this chapter can be applied to the development of financial trading systems in which there is no explicit step that forecasts fi-

nancial time series; rather, forecasts are formed implicitly by a controller in the form of an RMLP. We build here on the work of [Moody and Wu, 1997, Moody et al., 1998] in the following ways: (1) we utilize dynamic neural networks with internal states as controllers; (2) we demonstrate the application of the on-line, second-order EKF training methods to the synthesis of trading strategies; and (3) we provide details regarding training and network architectures.

This section consists of four parts. Section 1.6.1 introduces the problem of financial portfolio optimization and its profit performance function. Section 1.6.2 discusses general issues required for successful application of the EKF algorithm for training a neurocontroller as a trading system. Section 1.6.3 describes results of our simulations. We conclude with a few thoughts on possible extensions of the proposed approach in Section 1.6.4.

1.6.1 Financial Portfolio Optimization

For the sake of simplicity, we consider an example of two assets, one of which is risk-free and has a fixed return, such as a government bond, and the second of which can be characterized as risky (i.e., an asset whose temporal evolution appears to be largely stochastic but may have some deterministic component), such as a stock. We assume an objective of maximizing profitability with little regard for the risk incurred (however, risk can be addressed as shown below). Following [Moody and Wu, 1997, Moody et al., 1998], we consider a multiplicative model of profitability in which accumulated wealth is completely reinvested at each time period, where the control decision is the fraction of accumulated wealth to be invested in the risky asset, with the remainder invested in the risk-free asset. (Moody et al. develop a formalism that allows for multiple risky assets to be simultaneously considered.) In addition, the profit performance function includes a term that models the impact on profitability due to transaction costs. On the other hand, we will ignore tax implications and will assume that the trading strategy does not impact market dynamics (i.e., the evolution of the risky asset's price series is not affected by the trading system's actions).

We denote the values at time t of the risk-free bond and risky security as $z^f(t)$ and $z(t)$, respectively. Similarly, we define the instantaneous returns for these two assets as $r^f(t) = z^f(t)/z^f(t-1) - 1 = r^f$ and $r(t) = z(t)/z(t-1) - 1$, where the risk-free bond has a constant return and the risky security has a return governed by the evolution of its price series. At any given time, we make a decision, $u(t) \in [0, 1]$, that allocates a fraction of accumulated wealth to the risky security with the remaining fraction $1 - u(t)$ allocated to the risk-free asset. In addition, we assume that there is a cost associated with reallocation that is a function of the absolute difference between two successive allocations: $|u(t) - u(t-1)|$. Denoting the total return at time t by $R(t)$, the accumulated

wealth at time T is given by

$$\begin{aligned}\mathcal{W}(T) &= \mathcal{W}(0)\prod_{t=1}^{T}\{1+R(t)\} \\ &= \mathcal{W}(0)\prod_{t=1}^{T}\left\{1+(1-u(t-1))r^f+u(t-1)r(t)\right\} \\ &\quad \times\{1-\delta\,|u(t)-u(t-1)|\}\ , \end{aligned} \tag{1.64}$$

where we begin at time $t = 0$ with wealth $\mathcal{W}(0)$, and where the transaction cost rate is given by δ. Note that this profit model does not allow for short sales. It thus implies a relatively conservative strategy (a short sale would be characterized by values of control $u(t)$ less than zero).

This model has a number of interesting implications. First, the instantaneous total return $R(t)$ is a function of two successive controls, $u(t)$ and $u(t-1)$, due to the presence of trading costs (in the absence of trading costs, the instantaneous return $R(t)$ is only a function of the preceding control $u(t-1)$). This implies that any trading decision must explicitly take into account trading decisions made at preceding points in time; this is a form of friction, and makes portfolio optimization a problem of dynamic, nonlinear optimization. It is also noteworthy that, due to the assumption that the risky asset's dynamics are not affected by an individual trading system's actions, it is not necessary to train an ID network or develop some other analytical expression that performs one-time-step-ahead forecasts of the price series or returns, since these series can be considered to be exogenous to the system of interest. Instead, it is necessary only to understand how the trading system's actions affect the overall objective of maximizing wealth, and this is given by equation (1.64). Thus, this type of portfolio optimization can be viewed as a problem of open-loop, rather than closed-loop, control.

The overall system can be described in block-diagram form as shown in Figure 1.9. The wealth calculator takes as input the current return for the risky security, $r(t)$, as well as the control signal, $u(t)$, which determines how the simple portfolio is to be rebalanced, and computes the total accumulated wealth, $\mathcal{W}(t)$. We have assumed that the wealth calculator remembers the previous values of control, risky asset return, and wealth. In addition, the wealth calculator has knowledge of the risk-free asset's rate of return and the transaction cost rate. The trading system may be viewed as a dynamic system which processes a stream of returns for the risky asset to generate a stream of control signals; additional information such as the actual price or the change in wealth may also be useful as inputs. This trading system ultimately has embedded knowledge of both trading costs and the return rate of the risk-free asset. In addition, the trading system must have knowledge of its previous actions, and it must develop some predictive capability for the risky asset's price series. We utilize dynamic neural networks as controllers to address both the need to account for trading costs as well as to develop implicit forecasts for the risky asset's price movements.

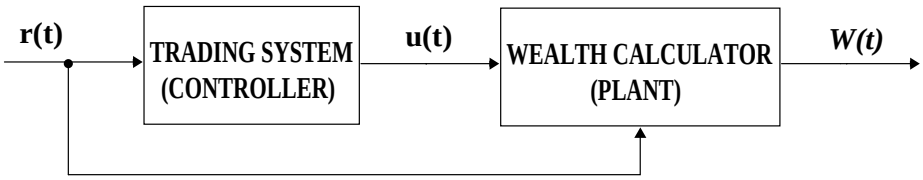


Figure 1.9: A block diagram representation of a simple, two-asset portfolio optimization system.

In the next sections we discuss our approach to trading with neurocontrollers. It is difficult to compare our approach with that of [Moody and Wu, 1997, Moody et al., 1998] in quantitative terms. It is not obvious as to whether Moody et al. are only using trading system networks with external input-to-output recurrence, or whether their systems explicitly utilize networks with internal dynamics. Furthermore, they provide little information as to what signals are used as inputs to the neural network trading system, and what temporal representation these signals take.

1.6.2 General Training Considerations

The optimization problem as formulated above poses certain difficulties for the EKF-based training methodology (see Section 1.3.2). First, while the primary objective is to maximize total wealth over time, the EKF-based methods typically apply only to the minimization of a sum of squared errors expressed as an objective function. Thus, it is necessary to convert the wealth maximization problem to some form of a minimization problem. A related difficulty is due to the form of the objective function: it is multiplicative, rather than additive, which precludes direct application of EKF training.

We address these two difficulties as follows. First, we transform the objective function by taking the logarithm of accumulated wealth and construct a logarithmic utility function (we assume with no loss of generality an initial wealth of $\mathcal{W}(0) = 1$):

$$\begin{aligned}
\tilde{\mathcal{W}}(T) &= \log \mathcal{W}(T) \\
&= \sum_{t=1}^{T} \Delta\tilde{\mathcal{W}}(t) \\
&= \sum_{t=1}^{T} \log\left\{1 + (1 - u(t-1))r^f + u(t-1)r(t)\right\} \\
&\quad + \log\left\{1 - \delta\left|u(t) - u(t-1)\right|\right\} , \qquad (1.65)
\end{aligned}$$

where we denote $\Delta\tilde{\mathcal{W}}(t)$ as the change in logarithmic wealth. Note that this transformation converts the multiplicative expression to one which is additive, which is more suitable for EKF training. In addition, this logarithmic utility

function, if used directly, introduces a degree of risk aversion [Moody and Wu, 1997] (the original wealth maximization formulation is risk-neutral).

This logarithmic transformation does not indicate directly how to convert the maximization of utility to a minimization of sum-of-squared errors for EKF-based training. However, this may be accomplished in one of two ways. First, we can associate with each EKF-based weight update step a small, constant and positive error signal which implies that regardless of how well or poorly the trading system is performing, it is desirable for the system to always do better; this mechanism acts as a reinforcement signal that is independent of the actual increment or decrement in accumulated wealth. Alternatively, the error signal can be defined as the difference between some maximum achievable change in logarithmic wealth, $\Delta\tilde{\mathcal{W}}_{\max}$, and the actual change in logarithmic wealth. In this case, the scalar error signal is still always positive, but increases as $\Delta\tilde{\mathcal{W}}(t)$ decreases. In either case, the derivative matrices $\mathbf{H}_i(t)$ (see Section 1.3.2) are constructed by computing the dynamic derivatives of $\Delta\tilde{\mathcal{W}}(t)$ with respect to the ith group of weight parameters for the trading system. Alternative means of constructing error or reinforcement signals can be considered as well. For example, if it is possible to determine that the trading system has performed properly in maximizing the change in logarithmic wealth at any given time, then it may be counterproductive to penalize this action by trying to adjust the trading system's parameters to perform better, when in fact it cannot; in this case, an error signal of zero may be appropriate.

1.6.3 Empirical Simulations

We employ the simulated time series studied by [Moody and Wu, 1997] to investigate the applicability of training recurrent neural networks with EKF methods for portfolio optimization. The price series is characterized as a random walk with an autoregressive trend process. This two-parameter model is given by

$$p(t) = p(t-1) + \beta(t-1) + k\lambda(t) \tag{1.66}$$

$$\beta(t) = \alpha\beta(t-1) + \gamma(t) \tag{1.67}$$

$$z(t) = \exp\left(\frac{p(t)}{1200}\right) \tag{1.68}$$

where $\lambda(t)$ and $\gamma(t)$ are two random normal processes with zero mean and unit variance, and where we have chosen $\alpha = 0.9$ and $k = 0.3$ for the simulation results described below.

We follow the convention established by [Moody and Wu, 1997] and consider the evolution and training of a trading strategy over $10,000$ time steps, where each time step represents an hourly interval in a 24-hour artificial market. We assume a trading cost rate of $\delta = 0.05$ and a rate of return for the risk-free asset of approximately 6.85% per annum ($r^f = 6 \times 10^{-6}$).

We consider the training of a neural network trader with three inputs: the first input is the value of the change in the logarithm of wealth from the previous time step, scaled by a factor of 50 (without scaling, the change in the logarithm

of wealth tends to be small); the second input is the instantaneous price of the risky asset; and the third input is the instantaneous value of the return of the risky asset, scaled by a factor of 100. The output of the trader is the trading signal $u(t)$, ranging between zero and unity. The output signal of the system we are attempting to control is the change in logarithmic wealth, scaled by a factor of 50. In EKF training, the error signal is given by $\xi(t) = 1 - 50\Delta\tilde{\mathcal{W}}(t)$, where we explicitly account for the scaling of the change in logarithmic wealth.

We consider the training of a recurrent controller with architecture denoted by 3-10R-5R-1U, where the output node is a unipolar sigmoid (1U in the notation). We assume at the beginning of a trajectory a naive trading strategy that sets $u(0) = 0.5$; this corresponds to initializing the controller with small random weight values. We train the controller with GEKF, where derivatives of change in scaled logarithmic wealth are computed with truncated backpropagation through time with a truncation depth of 10. Note that both the wealth calculator and the trading system are dynamic. A learning rate of $\eta = 0.01$ is employed throughout, and we use small constant process noise, $\mathbf{Q} = 10^{-8}\mathbf{I}$ (see Section 1.3.2).

We report here on a typical simulation run. We arbitrarily start with one unit of wealth, $\mathcal{W}(0) = 1$, and assume that the price series starts at a value of $z(0) = 1$. For each time step corresponding to the evolution of the price series we conduct one step of GEKF training.

Typical training results are shown in Figure 1.10. The bottom panel shows the price series in logarithmic form, which is one of the neurocontroller's inputs, in its raw form. The middle panel shows the evolution of the trading signal over the course of 10,000 time steps of training, when training is carried out at each time step. Note that the trading signal takes on values between zero and unity, but is not constrained to be saturated at either of these two extremes, and often takes on intermediate values. Finally, we plot the logarithm of accumulated wealth in the uppermost panel for three different training scenarios. In the lowest (dashed) trace, training was disabled at 2000 time steps, after which the controller was used without further update. Note that the logarithmic wealth basically remains unchanged after time step 2000, which was due to the fact that the neurocontroller was emitting a constant signal of $u(t) \approx 0$ for $t > 2000$. The middle trace demonstrates the evolution of logarithmic wealth when training is disabled at 4000 time steps. Unlike the previous case of disabling training at 2000 time steps, we note that there is appreciable growth in logarithmic wealth after 4000 time steps, which indicates that the neurocontroller has begun to implicitly model the underlying dynamics of the price series. Finally, the uppermost trace corresponds to the case in which training is carried out continuously; nearly identical behavior is observed when training is disabled at 7000 or more time steps.

It is noteworthy to compare the total accumulated wealth at 10,000 time steps to the value of the price series at the same point in time. Note that the price series ends up at a value of approximately 0.5 units of wealth, well below its initial value of one unit. On the other hand, the accumulated wealth at the end of 10,000 time steps is approximately 5.5 units. To put this result into context,

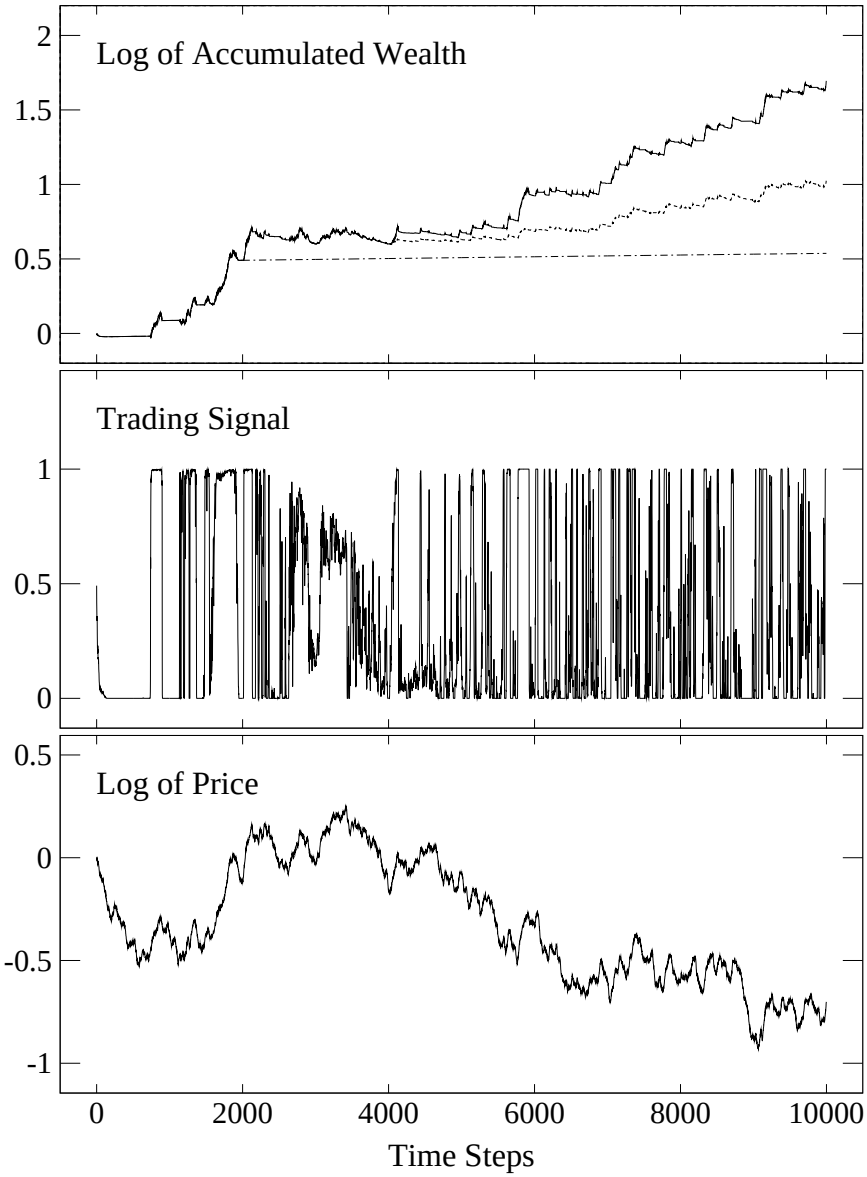


Figure 1.10: Typical results of training on the simulated price series. The price series and the accumulated wealth are plotted in the bottom and top panels, respectively, in a natural logarithmic scale. The middle panel shows the trading signal as the system is being trained. The top panel of wealth plots shows 3 traces; the bottom two correspond to disabling training at 2000 and 4000 time steps.

investing the initial unit of wealth exclusively in the risk-free asset would have resulted in an accumulated wealth of only 1.06 units after 10,000 time steps.

In Figure 1.11, we show at a finer scale the results between time steps 5500 and 6500. Note that the trading system appears to be able to effectively identify those periods of time when the simulated stock price starts to increase or decrease in value, and takes and maintains appropriate positions (e.g., the trading system invests in the risky asset shortly after it starts to increase in value and invests in the risk-free asset when the risky asset's value is declining). At times, the trading system takes intermediate positions, apparently reflecting its uncertainty in anticipated future movements of the price series. It is also noteworthy that the amount of observed switching in the trading signal is affected by the setting of the trading cost rate, where higher cost rates result in less aggressive strategies.

1.6.4 Possible Extensions

A number of useful extensions to this simple example can be easily developed. First, a more meaningful portfolio of more than a single traded asset should be considered. A general portfolio of N securities would require a neurocontroller with N outputs, where the outputs are normalized to add up to unity; Moody et al. [Moody and Wu, 1997, Moody et al., 1998] develop strategies for multiple asset portfolios and demonstrate the development of a neural network trader for a portfolio of three simulated assets, using the prices series defined above, but with different parameter settings for each of the price series.

While we have considered here the on-line adaptation of a trading strategy for a simulated price series with some deterministic component, a particularly useful extension would be to use multi-stream EKF training to develop a trading strategy off-line for many different forms of price series. In fact, one could choose to use recorded historical price series of a variety of different securities as a basis for a trading strategy. In addition, other information regarding *fundamentals* of the corporation issuing the stock could be considered as input to provide context; although this type of fundamental input would change infrequently, it could be expected to be quite different from one company to another.

[Moody and Wu, 1997, Moody et al., 1998] consider a variety of alternative optimization objective functions that provide a trade-off between risk and return. As a means of augmenting the formalism described here, one could have a multi-objective cost function that uses both change in logarithmic utility and portfolio volatility. This procedure would require a volatility "calculator" in addition to the wealth calculator; such a volatility calculator can be represented with recurrent neural networks. An off-line controller training strategy would use as input a reference signal for some maximum value of volatility that would be tolerated. Then, the cost function corresponding to the volatility term would only be invoked when the volatility of returns starts to exceed the reference signal (and perhaps only for down-side risk).

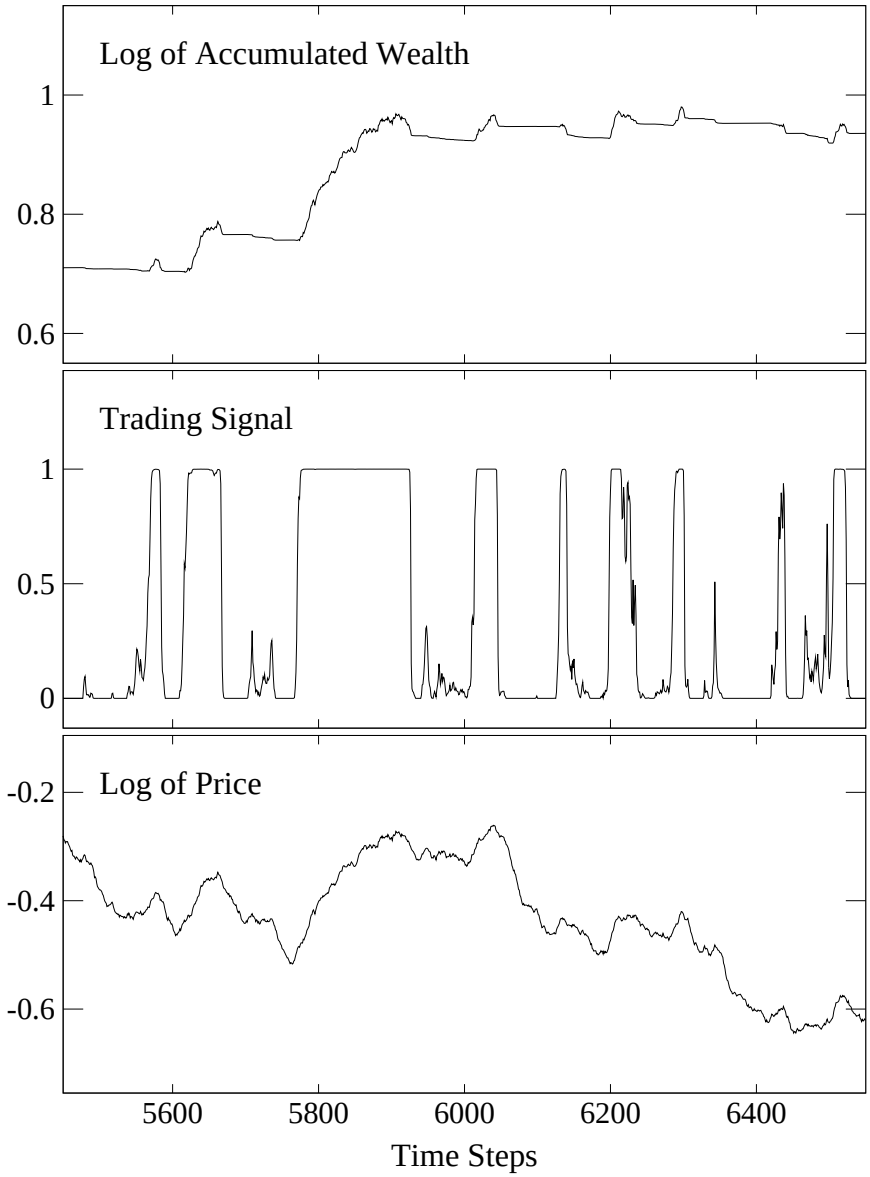


Figure 1.11: A section of 1000 points from the simulated price series shown in Figure 1.10 which demonstrates the dynamic nature of the trading system's actions: when the risky asset begins to appreciate in value, the trading system invests in that asset and properly maintains that position until the asset starts to appreciably decrease in value.

1.7 Conclusion

In this chapter, we have described our framework for effective application of dynamical neural networks in control. Essential elements of this framework were discussed. These included network execution, derivative calculations, and weight update mechanism. We emphasized that our approach to controller synthesis is fundamentally equivalent to training a heterogeneous recurrent neural network. We introduced the modular approach to make descriptions of neurocontrol system training more compact. We presented the case for multi-streaming and illustrated its effectiveness in an example application. While training for robustness was historically the first application of multi-stream EKF, other applications have been successfully attempted since (see [Feldkamp et al., 1998, Marko et al., 1996, Feldkamp et al., 1997, Feldkamp and Puskorius, 1997], [Feldkamp and Puskorius, 1998, Petrosian et al., 2000]).

In the first example, we utilized perhaps the simplest strategy for robustness training, i.e., choosing plants at random from the range of their parameters. One of our future directions is the exploration of a more directed selection scheme.

For this chapter, we adopted the setting of developing neurocontrollers in simulations, rather than through interaction with a physical system. Of course, a synthesized neurocontroller must be eventually deployed and tested on a real system. It is assumed that real system variations can be captured in the process of training for robustness (Section 1.5 of this chapter). Then a reasonable generalization may be warranted. Even so, the performance may turn out to be less than satisfactory, and some amount of on-line (after-deployment) training may be required. It is legitimate to ask whether our framework (or elements thereof) can be applied for on-line training. A comprehensive answer to this question could fill a separate chapter. Simply put, the answer is yes, but there are many issues to be addressed in order to make parts of the framework suitable for on-line training. Some of these issues (most notably, closed-loop stability) are not particular to neurocontrol systems, and apply to any adaptive control system. Others are specific to our framework (for example, further modifications required for multi-streaming). We refer an interested reader to [Puskorius et al., 1996] for a successful example of on-line controller training utilizing elements of the framework described.

We also demonstrated, through the second example, application of the neurcontroller training framework to an abstract financial, rather than physical, system. This example is contrived to illustrate that effective trading strategies can be directly inferred, provided that there is some degree of predictability in the underlying time series, even though an explicit step of time series prediction is never performed. Obviously, we do not expect real financial time series to exhibit the same level of predictability as is found in this example. Furthermore, we expect that successful application of these ideas for real financial systems is likely to be difficult.

Projects

1. Demonstrate (e.g., in computer simulations) that close approximation of the optimal solution in the linear quadratic regulation problem described on p. 205 of [White and Sofge, 1992] requires the use of $h \geq 1$ in BPTT(h).
2. Obtain backpropagation (dual) equations for the plant of Narendra and Mukhopadhyay.
3. Improve the robustness results of the recurrent controller of the first example (see Section 1.5) and report findings to the authors.
4. Apply the approach of Section 1.6 to other time series and synthesize a successful neural network trader. (Other training algorithms may be utilized.)
5. Apply the approach of Section 1.6 to the same time series, but sampled less frequently (e.g., once every 10 hours in the artificial market).
6. Implement the extensions proposed in Section 1.6.4

Bibliography

[Anderson and Moore, 1979] Anderson, B. D. O. and Moore, J. B. (1979). *Optimal Filtering*. Prentice Hall.

[Bishop, 1995] Bishop, C. (1995). *Neural Networks for Pattern Recognition*. Clarendon, Oxford.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

[Feldkamp and Prokhorov, 1998] Feldkamp, L. A. and Prokhorov, D. V. (1998). Phased backpropagation: a hybrid of bptt and temporal bp. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, volume III, pages 2262–2267, Anchorage, AK.

[Feldkamp et al., 1998] Feldkamp, L. A., Prokhorov, D. V., Eagen, C. F., and Yuan, F. (1998). Enhanced multi–stream kalman filter training for recurrent networks. In Suykens, J. and Vandewalle, J., editors, *Nonlinear Modeling: Advanced Black–Box Techniques*, pages 29–53. Kluwer Academic Publishers.

[Feldkamp and Puskorius, 1994] Feldkamp, L. A. and Puskorius, G. V. (1994). Training controllers for robustness: Multi-stream dekf. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 2377–2382, Orlando.

[Feldkamp and Puskorius, 1997] Feldkamp, L. A. and Puskorius, G. V. (1997). Fixed weight controller for multiple systems. In *Proceedings of the 1997 International Joint Conference on Neural Networks*, volume II, pages 773–778, Houston, TX.

[Feldkamp and Puskorius, 1998] Feldkamp, L. A. and Puskorius, G. V. (1998). A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering and classification. *Proceedings of the IEEE*, 86(11):2259–2277.

[Feldkamp et al., 1997] Feldkamp, L. A., Puskorius, G. V., and Moore, P. C. (1997). Adaptive behavior from fixed weight networks. *Information Sciences*, 98:217–235.

[Giles et al., 1990] Giles, C., Sun, G., Chen, H., Lee, Y., and Chen, D. (1990). Higher order recurrent networks & grammatical inference. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 380–387, San Mateo, CA. Morgan Kaufmann Publishers.

[Grossberg, 1982] Grossberg, S. (1982). *Studies of Mind and Brain: Neural Principles of Learning Perception, Development, Cognition, and Motor Control.* Reidell Press, Boston, MA.

[Haykin, 1996] Haykin, S. (1996). *Adaptive Filter Theory.* Prentice Hall.

[Haykin, 1999] Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 2nd edition.

[Jordan, 1986] Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitive Science Society*, pages 531–546. Lawrence Erlbaum.

[Lo, 1993] Lo, J. T.-H. (1993). System identification by recurrent multilayer perceptrons. In *Proceedings of the World Congress on Neural Networks*, volume IV, pages 589–600, Portland, OR.

[Marko et al., 1996] Marko, K. A., James, J. V., Feldkamp, T. M., Puskorius, G. V., Feldkamp, L. A., and Prokhorov, D. (1996). Training recurrent networks for classification: realization of automotive engine diagnostics. In *Proceedings of the World Congress on Neural Networks*, pages 845–850, San Diego.

[Moody and Wu, 1997] Moody, J. and Wu, L. (1997). *Optimization of Trading Systems and Portfilios*, pages 23–35. World Scientific.

[Moody et al., 1998] Moody, J., Wu, L., Liao, Y., and Saffell, M. (1998). Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting*, 17:441–470.

[Narendra and Mukhopadhyay, 1994] Narendra, K. S. and Mukhopadhyay, S. (1994). Adaptive control of nonlinear multivariable systems using neural networks. *Neural Networks*, 7(5):737–752.

[Narendra and Parthasarathy, 1990] Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1):4–27.

[Narendra and Parthasarathy, 1991] Narendra, K. S. and Parthasarathy, K. (1991). Gradient methods for the optimization of dynamical systems containing neural networks. *IEEE Transactions on Neural Networks*, 2(2):252–262.

[Petrosian et al., 2000] Petrosian, A., Prokhorov, D., Homan, R., Dasheiff, R., and Wunsch, D. (2000). Recurrent neural network based prediction of epileptic seizures in intra- and extracranial eeg. *Neurocomputing*, 30:201–218.

[Prokhorov and Feldkamp, 1997] Prokhorov, D. V. and Feldkamp, L. A. (1997). Primitive adaptive critics. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, volume IV, pages 2263–2267, Houston, TX.

[Prokhorov and Feldkamp, 1998] Prokhorov, D. V. and Feldkamp, L. A. (1998). Bayesian regularization in extended kalman filter training of neural networks. In *Proceedings of the 10th Yale Workshop on Adaptive Systems*, pages 77–82, New Haven, CT.

[Puskorius and Feldkamp, 1991] Puskorius, G. V. and Feldkamp, L. A. (1991). Decoupled extended kalman filter training of feedforward layered networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 771–777, Seattle, WA.

[Puskorius and Feldkamp, 1994] Puskorius, G. V. and Feldkamp, L. A. (1994). Neurocontrol of nonlinear dynamical systems with kalman filter trained recurrent networks. *IEEE Transactions on Neural Networks*, 5(2):279–297.

[Puskorius and Feldkamp, 1999] Puskorius, G. V. and Feldkamp, L. A. (1999). Roles of learning rates, artificial process noise and square root filtering for extended kalman filter training. In *Proceedings of the 1999 International Joint Conference on Neural Networks*, Washington D.C.

[Puskorius et al., 1996] Puskorius, G. V., Feldkamp, L. A., and Davis Jr., L. I. (1996). Dynamic neural network methods applied to on–vehicle idle speed control. *Proceedings of the IEEE*, 84(10):1407–1420.

[Sejnowski and Rosenberg, 1987] Sejnowski, T. J. and Rosenberg, C. R. (1987). Parallel networks that learn to pronounce english text. *Journal of Complex Systems*, 1:145–168.

[Singhal and Wu, 1989] Singhal, S. and Wu, L. (1989). Training multilayer perceptrons with the extended kalman algorithm. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 133–140. Morgan Kaufmann.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[White and Sofge, 1992] White, D. A. and Sofge, D. A., editors (1992.). *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold. See the example on p. 205.

[Williams and Peng, 1990] Williams, R. and Peng, J. (1990). An efficient gradient–based algorithm for on–line training of recurrent network trajectories. *Neural Computation*, 2:490–501.

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

[Williams and Zipser, 1995] Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Chauvin, Y. and Rumelhart, D. E., editors, *Back-propagation: Theory, Architectures and Applications*. Lawrence Erlbaum Associates. Also published as WILLIAMS-ZIPSER90A.

Glossary

Neurocontrol is a part of the field of neural networks dealing with theoretical and practical issues of the application of neural networks to control of various systems. It is usually assumed that a system to be controlled (or plant) can be modeled, e.g., with the help of an identification neural network.

Identification (ID) network is a component of the overall control system used for system modeling (identification). In indirect model-reference control with neural networks, it provides sensitivity signals necessary to train a controller network (or neurocontroller).

Model-reference control is a framework of the control theory which features a reference model. A simple block diagram of a model-reference control system is given in Figure 1.2 of Chapter 15. This framework was first adapted for neurocontrol by Narendra and Parthasarathy [Narendra and Parthasarathy, 1990].

Extended Kalman filter (EKF) recursion is a set of equations implementing the extended Kalman filter (EKF) algorithm. The EKF algorithm is rooted in the theory of optimal filtering. It can be successfully applied to training neural networks (see Section 1.3.2 of Chapter 15).

automated deduction systems

approximation capability

backpropagation through structure

BPTS *backpropagation through structure*

cascade-correlation networks

causality

computational power

connectionst structure processing

contour-tree

Index