HANDBOOK OF INTELLIGENT CONTROL

NEURAL, FUZZY, AND ADAPTIVE APPROACHES

Edited by
David A. White
Donald A. Sofge

1992

VNR VAN NOSTRAND REINHOLD
New York

10

NEURAL NETWORKS, SYSTEM IDENTIFICATION, AND CONTROL IN THE CHEMICAL PROCESS INDUSTRIES

Paul J. Werbos
NSF Program Director for Neuroengineering
Co-Director for Emerging Technologies Initiation

Thomas McAvoy and Ted Su
Department of Chemical Engineering, University of Maryland

10.1. INTRODUCTION AND SUMMARY

This chapter will discuss the application of artificial neural networks (ANNs) in the chemical process industries, with particular emphasis on new designs that could be useful in other applications (neural and nonneural) as well:

- The use of simple ANNs as the system identification component of model-based predictive control (MPC), a widely used scheme that optimizes over time subject to constraints.
- The use of "robust" or "parallel" training of ANNs used in system identification. In tests on real data from a wastewater treatment plant and an oil refinery, this has led to an orders-of-magnitude reduction in prediction error when compared with conventional maximum-likelihood approaches. Earlier tests on *nonneural* implementations were also promising. There are deep theoretical lessons involved, significant to all forms of ANN adaptation and system identification.
- Techniques for pruning ANNs (a task which has turned out to be crucial to ANN performance), techniques for using ANN-like techniques with *other* nonlinear specifications (for those who do *not* want to use ANNs as such) and techniques for using ANN models similar to nonlinear ARMA models or Kalman filtering systems—to be given in later sections.

ANNs are already in use in real-world applications by many of the major chemical manufacturers. Applications fielded to date mostly involve pattern recognition or soft sensing or feedforward control

systems, where safety and stability are not a major issue. There is substantial new research, and some application, in more advanced forms of system identification and control. Greater efficiency in chemical processing, through advanced control, could save millions of dollars and have a significant impact on major global environmental problems.

This chapter will begin with a broad summary of these applications. Then, in section 10.3, it will discuss the use of simple ANNs within model-based predictive control (MPC), and discuss some tests run on a *simulated* chemical plant at the University of Maryland [1]. Section 10.4 will discuss alternative ways of adapting ANNs for use in prediction, tested against *real-world* data. The theoretical issues discussed in section 10.4 apply to system identification in general; they are not unique to ANNs. Section 10.4 will also discuss alternative model specifications—neural and non-neural—which address such issues as measurement noise, slowly-varying system parameters, and memory of important but unobserved variables. Finally, the final sections of the chapter will provide the equations needed to implement various extensions of these methods; for example, section 10.5 will discuss a technique for pruning ANNs [2], and section 10.6 will contain a tutorial on how to use the chain rule for ordered derivatives [3, 4], so as to calculate gradients and Hessians at minimum cost in a parallelizable manner for *any* nonlinear dynamic system. Sections 10.7 and 10.8 will provide equations for options discussed in sections 10.3 and 10.4.

10.2. GENERAL BACKGROUND ON CHEMICAL APPLICATIONS

The Chemical Engineering Department at the University of Maryland, in cooperation with the NSF-sponsored Systems Research Center, has investigated a wide range of applications of artificial neural networks (ANNs). Because this work was successful, and also because of substantial industrial interest, McAvoy founded a Neural Network Club in 1990 that now has over twenty-five dues-paying corporate members, representing Fortune 500 corporations. Many of these companies have achieved net savings already, through existing applications of ANNs, which range from hundreds of thousands to millions of dollars. At a recent public talk [5], a representative from DuPont stated that his company is already into the "third generation" of ANN applications, reaching out to dozens or even hundreds of sites within the company.

The applications of ANNs to date at the Neural Network Club are probably representative of those in the industry:

1. Sensor interpretation (biosensors) [6]
2. Nonlinear dynamic modeling [7,8]
3. Modeling (or "cloning" of human operators/experts) [9]
4. Nonlinear steady-state modeling
5. Knowledge extraction [10]
6. Smart optimization [11]
7. Fault detection [12]

In addition, a technique has been developed to remove unnecessary connections from ANNs [2] (see section 10.5), using a modified objective function. Techniques of this sort are important to the success of many applications involving backpropagation.

The sensor-interpretation example [6] is probably typical of the easy, first-generation applications that are now widespread and still expanding. In this example, McAvoy et al. were basically trying to build a "soft sensor" capable of monitoring the concentration of desirable products in real time in a chemical plant. Traditionally, the process industries have relied on simple measurements such as flow, pressure, and temperature; however, to achieve a higher level of efficiency, and to control more complex systems such as bioreactors, it is important to measure concentrations as well.

Chemical engineers have known for decades how to measure chemical concentrations *in the laboratory, if* one is willing to wait for a few hours or a few days. The challenge here lies in *real-time* sensing. We do have real-time measurements available—such as fluorescence spectra—which *implicitly* tell us what the concentrations are; the problem is to learn the nonlinear mapping from the *spectral information* to *actual concentrations*. In their experiment, McAvoy et al. did exactly that, by putting together thirty-three samples of various mixtures of amino acids, and using a simple ANN to predict the corresponding concentrations of fluorophores. They achieved an average absolute prediction error of 5.2% in predicting the mole fractions, using an ANN, versus 11.7% when they used partial least squares (PLS), the best conventional alternative [13]. A larger training database should make the method more accurate.

ANNs for pattern recognition have also been used to detect hot spots in the continuous casting of steel [14]. In this application, ANNs led to a great increase in accuracy over more expensive, conventional methods. The system has been installed in several plants of the Nippon Steel Company and is scheduled for use in all of them.

Nonlinear dynamic modeling—like smart sensing—is crucial to real-time control. For that reason, this chapter will describe our recent experience in that area in some detail. First-principle models are possible in the chemical industry, as in other industries; however, the sheer complexity of chemical plants and the presence of unknown secondary reactions usually make such models extremely expensive, unreliable, and specific to only one or two plants. For this reason, *empirically* based models—either linear models or ANN models—can be of great value. ANN models can be used either as alternatives to first-principle models or as supplements, to predict the errors of the first-principle models.

The modeling of human operators can also be very useful [9]. The difference between the best human operators and the average is often worth large amounts of money, because of differences in plant efficiency during plant change-over and the like. The techniques required to model human operators with ANNs are no different, in principle, from those required to model other dynamic systems. This approach is discussed further in the section on supervised control in Chapter 3.

ANNs have been used at Maryland and elsewhere to extract knowledge that is then used in more conventional AI control systems [10]. There are probably dozens of ways to combine ANNs and AI techniques, useful in different applications, especially if the AI systems are based on differentiable functions (as in fuzzy logic). Many of the best examples of such combinations come from Japan [15].

More effective optimization—both in the design of chemical plants [11] and in real-time optimization—will probably turn out to be the most important application in the long term. As an example, imagine a petrochemical plant that produces a billion dollars worth of product per year, running at a ninety-eight percent level of efficiency. Such efficiencies are quite typical. The remaining *in*efficien-

cies are typically due to the difficulty of optimization during transient or changing conditions, when process parameters change in an unplanned or nonlinear manner. Good nonlinear optimization might well raise this up to ninety-nine percent efficiency, thereby saving the company $10 million per year.

More importantly, a rise in efficiency from ninety-eight percent to ninety-nine percent typically cuts the *unwanted waste products* in half, from two percent to one percent. It is not obvious what the larger social implications of this may be, because there are so many different kinds of plants and waste products; however, as a general rule, there *is* a correlation between inefficiency and waste, and there *is* reason to believe that such improvements could have a substantial impact on environmental problems. As a further example, a representative of General Electric (GE) has told us that more efficient, pattern-recognition-based control systems might well be able to reduce the emissions of NOx by a factor of ten, in gas turbines and boilers, even as a retrofit to well-designed modern systems. These kinds of environmental benefits address central problems that would otherwise be very difficult to deal with in a comprehensive, long-term energy and environmental policy [55].

Better control may be crucial, as well, to the *feasibility* of using certain advanced production processes, such as the biological processes used in biotechnology and wastewater treatment plants. The potential markets for such processes are very large.

Better diagnostics are important, of course, to improved safety and to cost-effective maintenance that can minimize the need for very expensive replacement investments. A wide variety of techniques has been used at the University of Maryland and elsewhere [12,14,16]. The issue of safety will be a crucial factor in the use of novel control designs in the chemical industry.

10.3. MODEL-BASED PREDICTIVE CONTROL WITH ANNS

Model-based predictive control (MPC) has been used very widely in the chemical and petroleum industries. Because of the difficulties of using or obtaining first-principle models (mentioned above), the usual procedure is to develop a *linear* model of the plant, based on empirical data, and to use that model within an optimization routine. Our goal has been to use the same empirical data, instead, to develop an ANN model of the same plant. This is almost as easy as developing the linear model; however, because ANNs can learn to represent any well-behaved nonlinear function [17,18,19], they should be able to represent the plant more accurately, and thereby lead to better performance.

The simulations described here were first reported by Saint-Donat, Bhat, and McAvoy [1] at Maryland. Saint-Donat has since returned to the ATOCHEM company of France, and Bhat has gone to Texaco.

Model-based control systems have been used before in the neural network community. For example, see the section on the "backpropagation of utility" in Chapters 3 and 13. The method used by Saint-Donat et al. is technically a form of the backpropagation of utility, but it has two special features that it shares with other forms of MPC:

- The present and future control signals are output *directly* from the optimizer, not from an ANN or other parametrized controller; in other words, the system solves for an optimal future schedule of actions, rather than an optimal set of controller parameters.
- The optimizer takes account, explicitly, of system constraints, which typically involve constraints on flow rates, temperature, and pressure.

This method is parallelizable, and the major pieces of it could be implemented on chips; therefore, unlike traditional MPC, it has serious promise as a real-time optimal control system.

This section will begin with a more detailed description of the task that Saint-Donat et al. addressed in simulation. Next, it will describe the architecture used to perform the task. It will conclude with a discussion of the observed performance.

10.3.1. The Task: A pH CSTR Reactor

The task is to control the level of acidity (pH) in a continuously stirred tank reactor (CSTR), illustrated in Figure 10.1. We assume a constant input flow of acetic acid. Our goal is to adjust the flow of sodium hydroxide (NaOH) so as to "neutralize" the acid—to achieve a desired setpoint for the pH. To test the control system, Saint-Donat et al. evaluated its ability to cope with a change in setpoints. In the initial tests, they changed the setpoint from 6.5 to 6.3. In later tests, they changed it from 7.2 to 7.0.

At any given time, t, we have information on the pH (which we observe) and the flow rate (which we chose), both for times $t-3$, $t-2$, and $t-1$. We also observe the pH at time t. Our goal is to decide on the flow rate for times t through $t+7$. The flow rates for times $t+8$ through $t+30$ will be set equal to the flow rate for time $t+7$; this is done to stabilize the system, to prevent "overoptimization," and is standard in MPC.

To be more precise, our goal at any time t is to pick the actions ($F_2(t)$ through $F_2(t+5)$), which will minimize the tracking error:

$$U = \sum_{k=1}^{30} (pH(t+k) - pH^*(t+k))^2, \tag{1}$$

where pH is the actual pH and pH^* is the desired pH, subject to four constraints that apply at all times:

$$pH_{\min} \leq pH \leq pH_{\max} \tag{2}$$

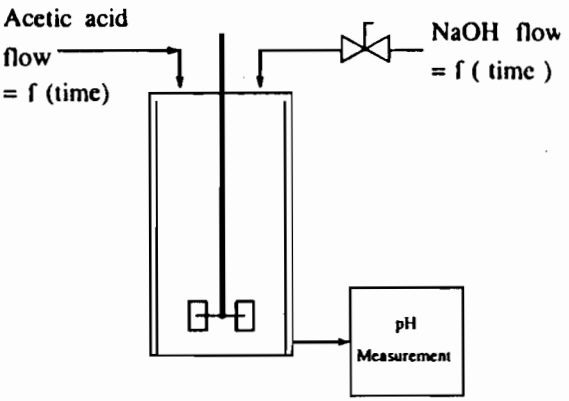


Figure 10.1 Case study (pH in a stirred tank reactor)

$$F_{2\min} \leq F_2 \leq F_{2\max}. \tag{3}$$

Tracking error is denoted by "U," rather than "E," because it is really a measure of utility or disutility—a measure of the desirability of the actual state of the plant; the letter "E" is reserved for prediction errors or classification errors or the like. Our control system as a whole is designed to minimize U, but it will contain a system identification component trained to minimize E.

The simulation was based on equations taken from [7]. By writing material balances on $Na^+$and total acetate ($HAC + AC^-$), and assuming that acid-base equilibria and electroneutrality relationships hold, one gets:

Total Acetate Balance:

$$\xi = [HAC] + [AC^-] \tag{4}$$

$$V\frac{d\xi}{dt} = F_1C_1 - (F_1 + F_2)\xi \tag{5}$$

Sodium Ion Balance:

$$\zeta = [NA^+] \tag{6}$$

$$V\frac{d\zeta}{dt} = F_2C_2 - (F_1 + F_2)\zeta \tag{7}$$

Electroneutrality:

$$\zeta + [H^-] = K_w/[H^-] + K_a[HAC]/[H^-] \tag{8}$$

F_1 and F_2 are the acid and base flow rates, respectively. In our case, F_2 is the manipulated variable. C_1 and C_2 are the concentrations of the acids and bases, respectively, in the input stream. V is the volume of the reactor. The parameters used in this simulation are given in Table 10.1.

Table 10.1

CSTR Parameters Used	Value
Volume of Tank	1000 lit.
Flow rate of HAC	81 lit./min.
Steady state flow rate of NaOH	515 lit./min.
Steady state pH	7
Concentration of HAC in F1	0.3178 mol./lit.
Concentration of NaOH in F2	0.05 mol./lit.

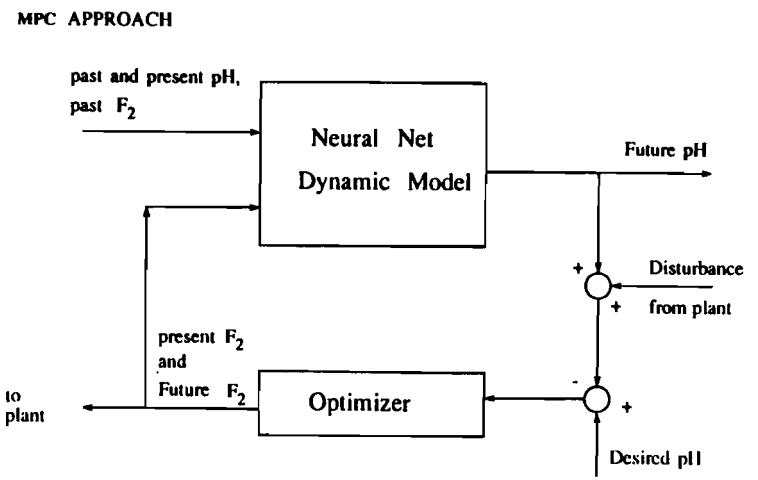


Figure 10.2 Neural net based control.

10.3.2. Architecture Used to Solve the Problem

The architecture used to solve this problem is shown in Figure 10.2. First of all, an ANN was adapted or fitted to serve as a dynamic model of the process to be controlled. Then, in real-time control, the ANN was held fixed, except for a constant bias term that was continually updated. At each time t, the optimizer looked ahead thirty steps and planned a series of actions for thirty time-steps ahead; however, it actually took only one step, and then—at time $t + 1$—it recalculated everything. This section will describe these steps in more detail.

10.3.2.1. Adapting the ANN The ANN was an ordinary three-layer multilayer perceptron (MLP), illustrated in Figure 10.3. In general, this kind of network inputs a vector of inputs, $\boldsymbol{X}(t)$, made up of components $X_1(t)$ through $X_m(t)$. It outputs a vector $\boldsymbol{Y}(t)$, made up of components $Y_1(t)$ through $Y_n(t)$. It obeys the following system of equations at each time t:

$$v_j^- = W_{j0}^- + \sum_{k=1}^{m} W_{jk}^- X_k(t) \qquad 1 \le j \le h \tag{9}$$

$$x_j^- = s(v_j^-) \triangleq 1/(1 + e^{-v_j^-}) \qquad 1 \le j \le h \tag{10}$$

$$v_i^+ = W_{i0}^+ + \sum_{j=1}^{h} W_{ij}^+ x_i^- \qquad 1 \le i \le n \tag{11}$$

$$\hat{Y}_i(t) = x_i^+ = s(v_i^+) \triangleq 1/(1 + e^{-v_i^+}), \qquad 1 \le i \le n \tag{12}$$

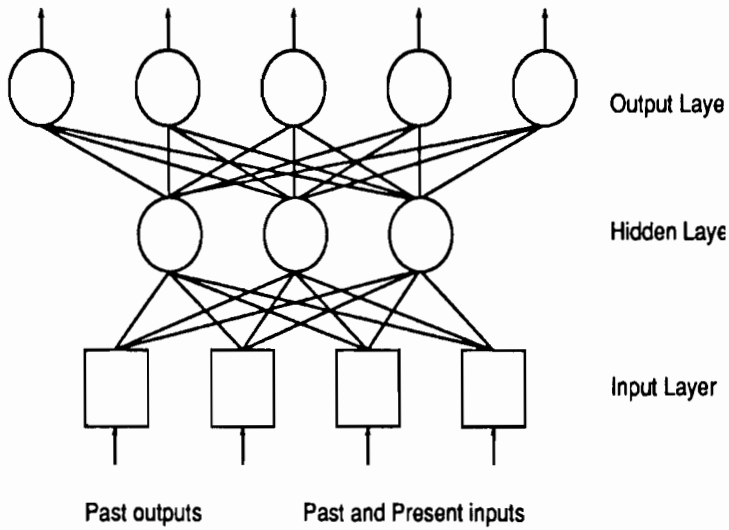


Figure 10.3 Multilayer perceptron.

where h represents the size of the hidden layer, where $s()$ is usually called "the sigmoid function," where the matrices W^+ and W^- are the "weights" or parameters to be adapted, where v^- and v^+ represent the levels of voltage stimulating the neurons on the hidden and output layers, respectively, and where x^- and x^+ represent the intensity of the outputs of those neurons.

In this application, Saint-Donat et al. used a net with seven inputs (i.e., $m = 7$), four hidden units ($h = 4$), and one output or target variable ($n = 1$). At each time t, the input vector, $X(t)$, consists of the following seven numbers: $pH(t-3)$, $pH(t-2)$, $pH(t-1)$, $F_2(t-3)$, $F_2(t-2)$, $F_2(t-1)$, *and* $F_2(t)$. The output, Y_1, was used to predict $pH(t)$. In actuality, the variable $F_2(t)$ had little effect on the forecast of $pH(t)$, because of deadtime effects, and was deleted from subsequent tests; however, this chapter will assume its inclusion, for the sake of generality. In earlier experiments [7], ANNs had been developed to predict $pH(t+1)$, $pH(t+2)$, and $pH(t+3)$ all together as outputs of the network; however, those ANNs were not used in this application. Figure 10.5 illustrates the fit between predicted pH and actual pH in some of the earlier work [7].

In order to adapt this network—i.e., to estimate the optimal values of the weights W_{ij}^- and W_{ij}^+—Saint-Donat, *et al.* performed two tasks: (1) they developed a database of simulated $X(t)$ and $pH(t)$ for many times t; (2) they used the well-known technique called backpropagation to adapt the weights, so as to minimize the error of the network in predicting $pH(t)$ over that database.

The technique for developing the database was taken from Bhat, *et al.* [7]. Basically, a pseudo-random binary signal (PRBS) was added to a steady-state signal for the manipulated variable, and equations 4 through 8 were used to forecast the resulting flows. This led to the time series for flow rate (F_2) and pH shown in Figure 10.4. This continuous time series was read off at regular time intervals, yielding the database for $X(t)$ and $pH(t)$ across different values of t.

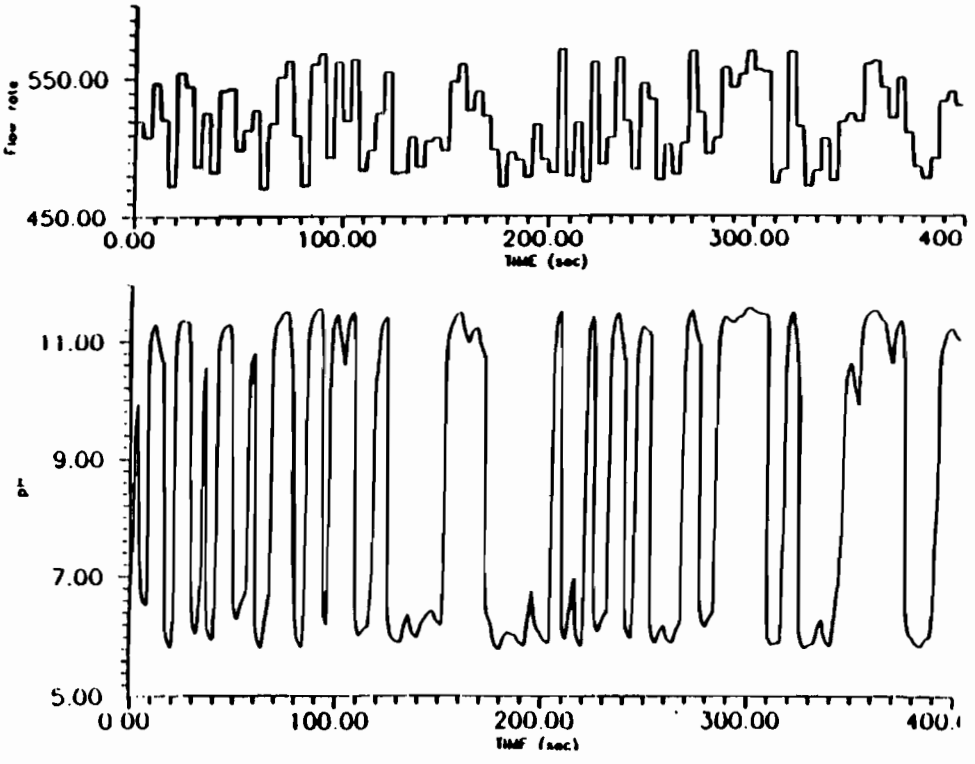


Figure 10.4 Case study: pH in a C.S.T.R. (Acid flow rate constant and NaOH flow rate varying.)

The technique used to adapt the weights was basic backpropagation (BEP)—a specialized use of backpropagation first proposed by Werbos [20–22] and later popularized by Rumelhart [23]. In this application, Saint-Donat et al. cycled through each of the $(X(t), pH(t))$ pairs in the database, performing the following sequence of steps for each pair t:

1. Use equations 9 through 12 to calculate a forecast, $\hat{Y}(t)$, for $pH(t)$. We will denote this forecast as $\hat{pH}(t)$.
2. Calculate the error of this forecast:

$$E(t) = \tfrac{1}{2}(\hat{pH}(t) - pH(t))^2 \tag{13}$$

3. Update the weights by the equations:

$$\text{new } W_{ij}^{\dagger} = \text{old } W_{ij}^{\dagger} - \textit{learning\_rate} * \frac{\partial E(t)}{\partial W_{ij}^{\dagger}} \tag{14}$$

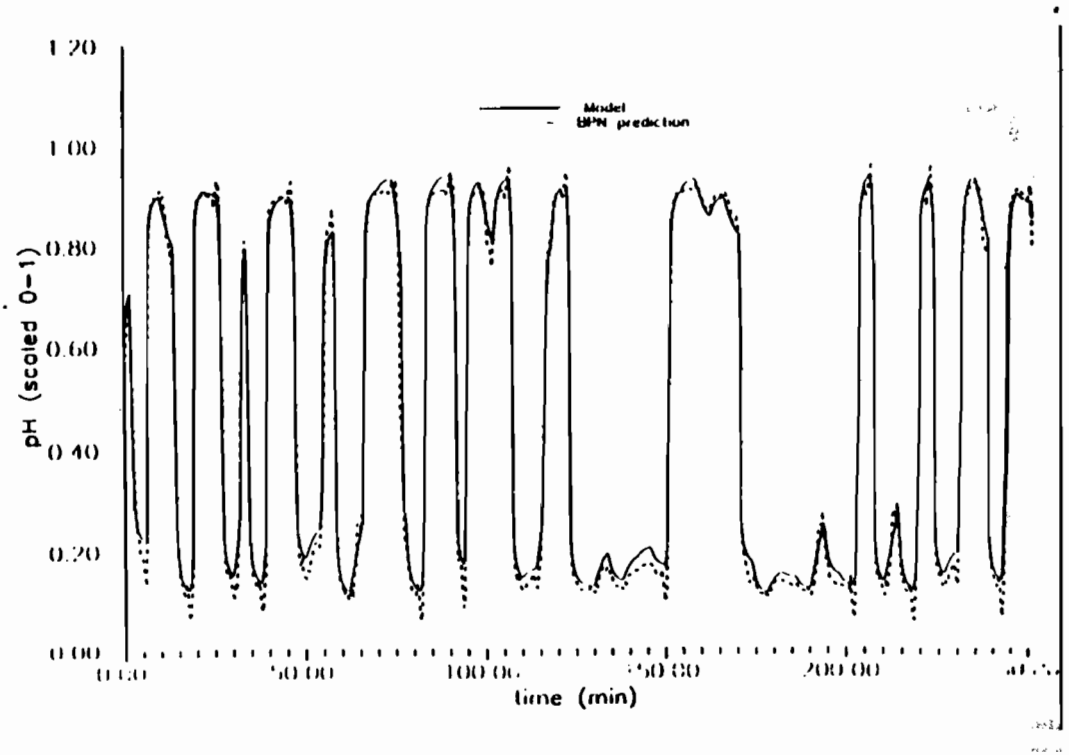


Figure 10.5 Prediction on training data (st. st. pH = 9).

$$\text{new } W_{ij}^{-} = \text{old } W_{ij}^{-} - learning\_rate * \frac{\partial E(t)}{\partial W_{ij}^{-}},$$

where the constant *learning_rate* was set as in [7]. They cycled through all of the $(X(t), pH(t))$ pairs, over and over, until the weights converged.

For the simple architecture shown in equations 9 through 12, it happens that:

$$\frac{\partial E(t)}{\partial W_{ij}^{+}} = (\hat{Y}_i(t) - Y_i(t)) * s'(v_i^{+}) * x_j^{-} \tag{15}$$

and

$$\frac{\partial E(t)}{\partial W_{jk}^{-}} = \left(\sum_{i=1}^{n} (\hat{Y}_i(t) - Y_i(t)) * s'(v_i^{+}) \right) * W_{ij}^{+} s'(v_j^{-}) X_k(t), \tag{16}$$

where the sigmoid function $s(\,)$ has the property:

$$s'(v) = s(v)(1 - s(v)). \tag{17}$$

In this application, $n = 1$ and $Y_1(t)$ is just $pH(t)$; this simplifies the calculations considerably.

Backpropagation, in the most general sense, refers to a general algorithm for calculating derivatives efficiently through a large sparse system, to be described in section 10.6. The efficient implementation of equations 16 and 17 is simply one example of the applications of that algorithm. Many authors have discussed alternatives to equation 14 which can speed up convergence [21,24,25]

10.3.2.2. Adapting the ANN Bias in Real Time Once the ANN was adapted off-line, as discussed in section 10.3.2.1., the weights were frozen *throughout* the control simulations, with one exception. At each time t, the forecast of $pH(t)$ actually used was:

$$pH(t) \text{ forecast} = \hat{pH}(t) + bias, \tag{18}$$

where $\hat{pH}(t)$ was the forecast from the ANN and *bias* was calculated from the previous time-step:

$$bias = pH(t-1) - \hat{pH}(t-1). \tag{19}$$

Furthermore, this same value of *bias* was used in predicting $pH(t + k)$, for $k = 1$ through 30, as required by the control system (optimizer) at time t. The bias term accounts for plant/model mismatch and unmeasured disturbances. By using equation 19, it can be shown that perfect setpoint tracking can be achieved in spite of these effects.

10.3.2.3. Solving the Optimization Problem in Real Time At each time t, the optimization routine solved for values of $F_2(t)$ through $F_2(t + 7)$ to maximize:

$$U = \sum_{k=1}^{30} (pH(t+k) - pH^*(t+k))^2,$$

subject to the constraints:

$$pH_{\min} \le pH \le pH_{\max}$$
$$F_2\min \le F_2 \le F_2\max$$
$$F_2(t+k) = F_2(t+7) \quad \text{for } k > 7,$$

as given in equations 1 through 3, assuming that $pH(t + k)$ is given as in equation 18.

Actually, equation 18 only tells us how to predict $pH(t)$ when we know $F_2(t)$ and prior information; Figure 10.6 shows the general approach used to "chain" the ANN to produce all the forecasts we need, from $pH(t)$ through to $pH(t + 30)$. (Some of these figures have minor differences from the nets used here, because they came from earlier work.)

This optimization problem is a straightforward problem in nonlinear programming. To solve this problem, Saint-Donat et al. [1] used a feasible sequential quadratic programming (FSQP) program developed through research at the University of Maryland [26,27]. This program can:

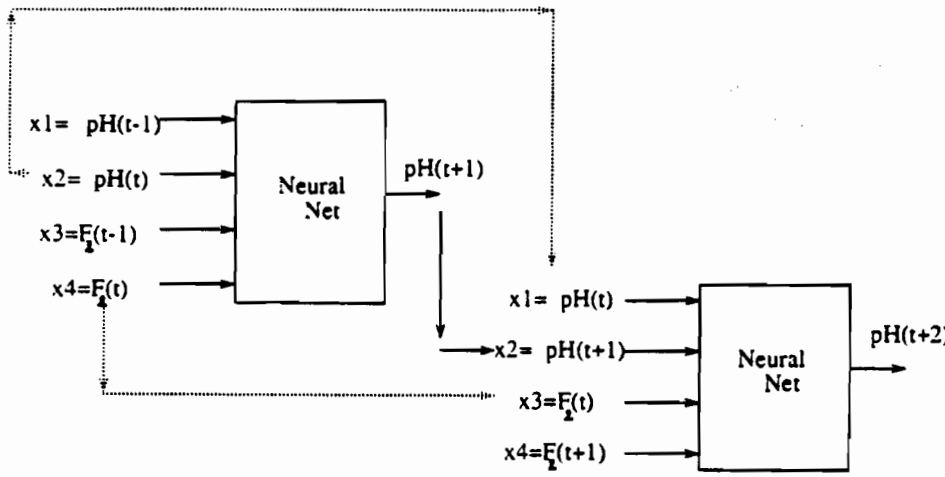


Figure 10.6 Simple example.

- Minimize any smooth nonlinear function
- Deal with any kind of nonlinear equality and/or inequality constraints on the manipulated variables
- Generate only feasible iterates (i.e., meet all the constraints even before convergence to the final solution)

This last feature will be especially important if we actually try to apply this approach in a real plant. In a real plant, one can never be certain that the FSQP will find the optimal solution quickly enough, if we require it to stop its calculations at the end of each sampling cycle. This could become a serious problem for complex plants. However, the feasibility property guarantees that the current iteration of FSQP will still give us acceptable values for the manipulated variables, even if they are not quite fully optimal. The feasibility feature has another advantage, which may help in speeding up convergence: By keeping the iteration out of the unacceptable range, it also keeps it within the original training range of the ANN; iterations outside that range might have confused the iterative process, because predictions from the ANN are far less reliable outside of the original training range.

In the future, there is reason to hope that this algorithm can be implemented in a fully parallel form, on a chip—enabling true real-time applications. ANN calculations can certainly be done on a chip. Even within the optimization algorithm, a major part of the computation concerns the gradient of the Hamiltonian function for the problem, given by:

$$H = U + \sum_{k=0}^{30} \left(\lambda_k^{(1)}(pH(t+k) - pH_{\min}) + \lambda_k^{(2)}(pH_{\max} - pH(t+k)) \right) \tag{20}$$

$$+ \sum_{k=0}^{30} \left(\lambda_k^{(3)}(F_2(t+k) - F_{2\min}) + \lambda_k^{(4)}(F_{2\max} - F_2(t+k)) \right)$$

where the λ's are Lagrange multipliers. In the special case where $pH(t + k)$ is predicted by an ANN, we can parallelize the calculation of this gradient, simply by using backpropagation through time. This was actually done successfully in the simulations by Saint-Donat et al. In fact, backpropagation can also be used to calculate the Hessian of the Hamiltonian—which can also be used in FSQP—in a highly efficient way. section 10.7 will provide the details of how to do both things, based directly on the chain rule for ordered derivatives.

Additional speedup results from using the values of $F_2(t + 1)$ through $F_2(t + 30)$, calculated at time t, as *starting* values for the optimizer at time $t + 1$. Deeper analysis of these Lagrange multipliers may also be useful. (However, users of methods like differential dynamic programming or DHP should be warned that these Lagrange multipliers are quite different, because the equations used to predict $pH(t + k)$ are not treated as constraints themselves.)

10.3.3. Simulation Results and Conclusions

Before testing the overall control scheme, Saint-Donat et al. conducted a brief test of the steady-state predictive abilities of the ANN. Figure 10.7 shows a comparison of the steady-state pH versus the steady-state flow of NaOH into the CSTR for a constant inflow of acid. As can be seen, the ANN achieves a reasonably accurate model of the highly nonlinear steady-state characteristics of the CSTR.

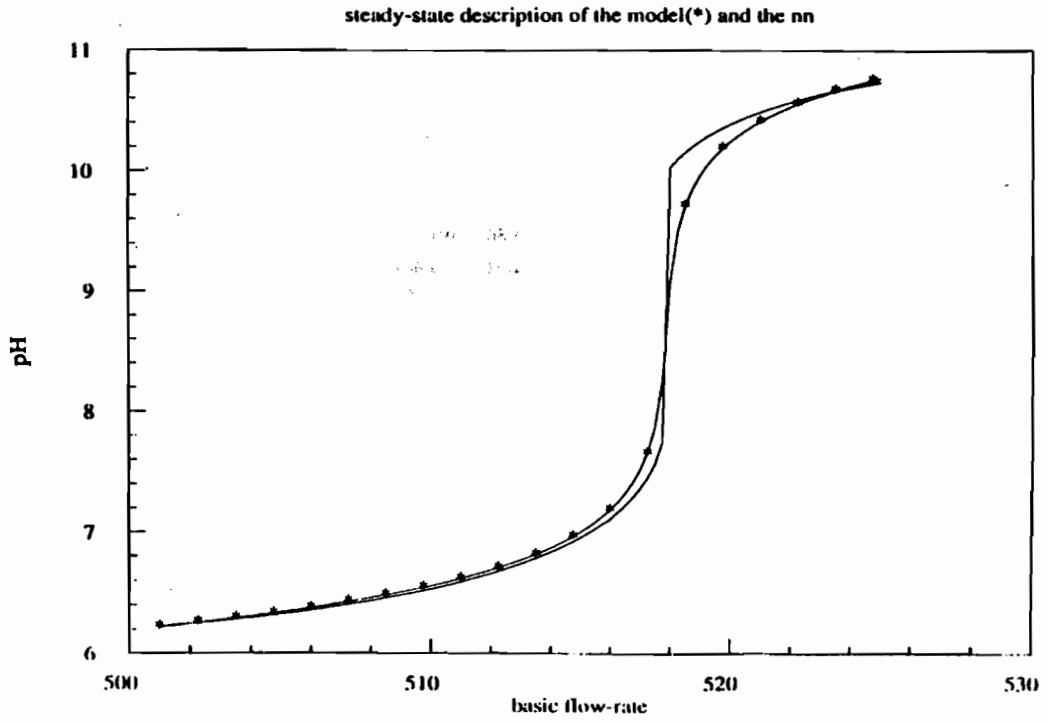


Figure 10.7 Steady state description of the model(*) and the nn.

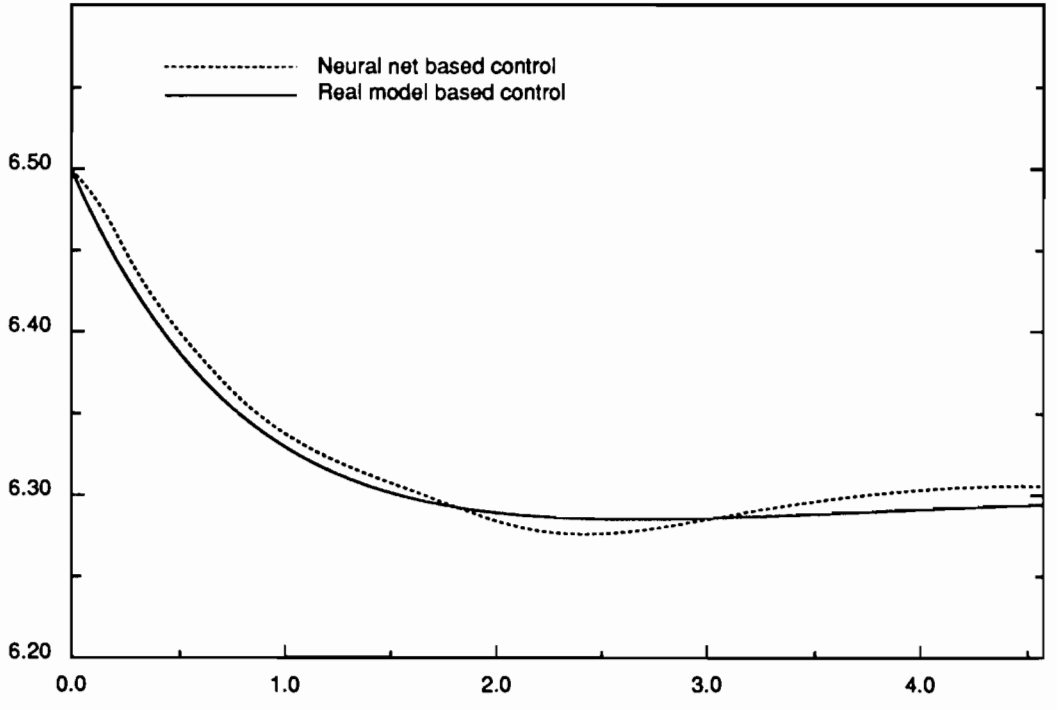


Figure 10.8 Controlled set-point change: 6.5 → 6.3.

To test the model-based control approach, the pH setpoint was initially set to 6.50. Then the setpoint was changed to 6.3. Figure 10.8 shows the actual change in acidity achieved by the neural net control system. Also shown is the response achieved when the real model is used in place of the ANN in Figure 10.2. As can be seen, the response achieved with the ANN is close to that achieved using the real model. In both cases, the new setpoint is not reached immediately, because the optimization hits constraints on the allowable flow. It is clear in this case that the ANN has captured the dynamics of the process. There was no need for first-principle modeling; the control architecture shown in Figure 10.2 does not need any prior knowledge about the plant. The ANN, although trained on a very wide region, is able to predict very accurately the dynamics of the pH system in this region between 6.5 and 6.3.

To provide a more challenging problem, the pH setpoint was initially set at 7.2. Then the setpoint was changed to 7.0. In this pH region, the steady-state pH gain to basic flow (i.e., the derivative of pH with respect to flow) changes by a factor of 2.8, as Figure 10.7 illustrates. Figure 10.9 shows the setpoint responses achieved by the ANN control system. Also shown is the response achieved when the real model is used in place of the ANN in Figure 10.2. As can be seen, the ANN-controlled response is slower than that of the real model and its undershoot is higher. It should be noted that the ANN was trained on a database where the pH varied between 6 and 11. If the ANN were trained on

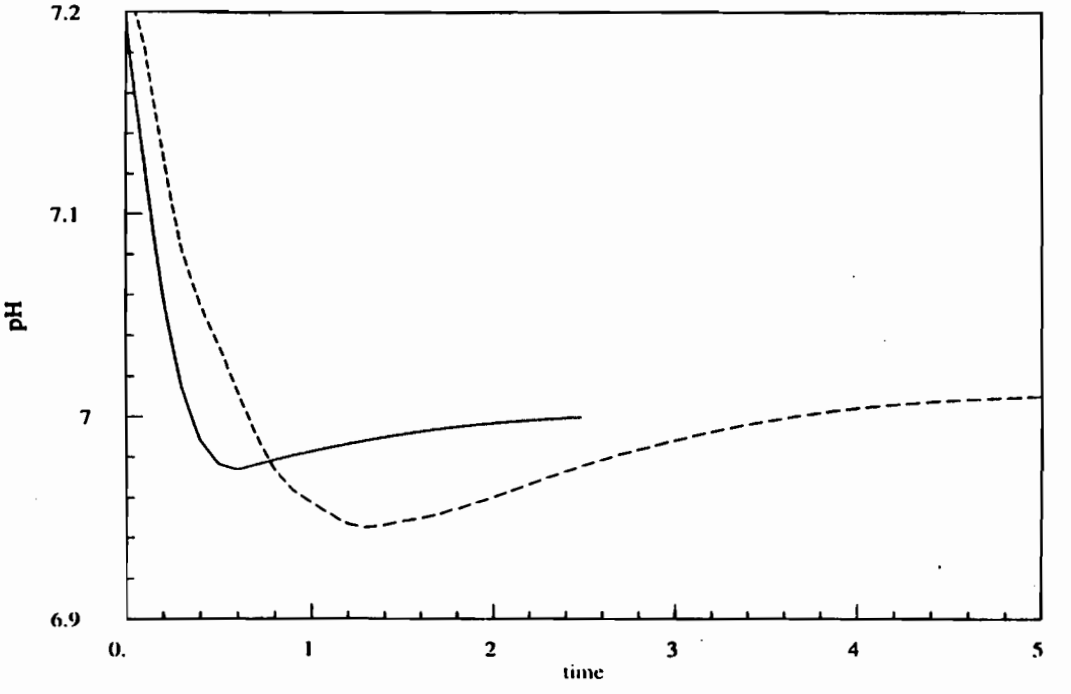


Figure 10.9 Controlled set-point change: 7.2 → 7.0.

data in the pH range 7.0 to 7.2, a more accurate model would be achieved. (This also suggests the possibility of something like neural gain scheduling, as a step up from the usual linear gain scheduling.) Although the process gain changes by a factor of 2.8 in the region between pH equal 7.2 to 7.0, the neural net is able to cope with this change effectively. Overall, the results shown in Figure 10.9 are encouraging, since the nonlinear neural net model was developed using only process data.

As can be seen, the ANN model used in a model predictive control architecture produces a response in certain regions that is slower than that produced by the real model. The question arises as to how this discrepancy can be explained. Figure 10.10 shows the derivative of the tracking error U with respect to the first future move of the manipulated variable (F_2). For each flow rate on the X-axis, the steady state is assumed to be reached. Tracking errors were calculated based on a setpoint of 7.0, which corresponds to an NaOH flow of 515 liters per minute. The ANN does a good job in most of the cases but has some problems in modeling the dynamics near pH = 7. This case study is particularly nonlinear, with the classic pH jump near 7. Given that the control architecture is tested in this region of pH near 7, results like those in Figures 10.9 and 10.10 are easy to explain. These results do not represent the limit of our system to control a plant, but rather the limits of this ANN in modeling velocity information. The ANN was trained on real input-output data, and was asked after convergence, not only to reproduce that data, but also to provide the first and eventually the second derivative

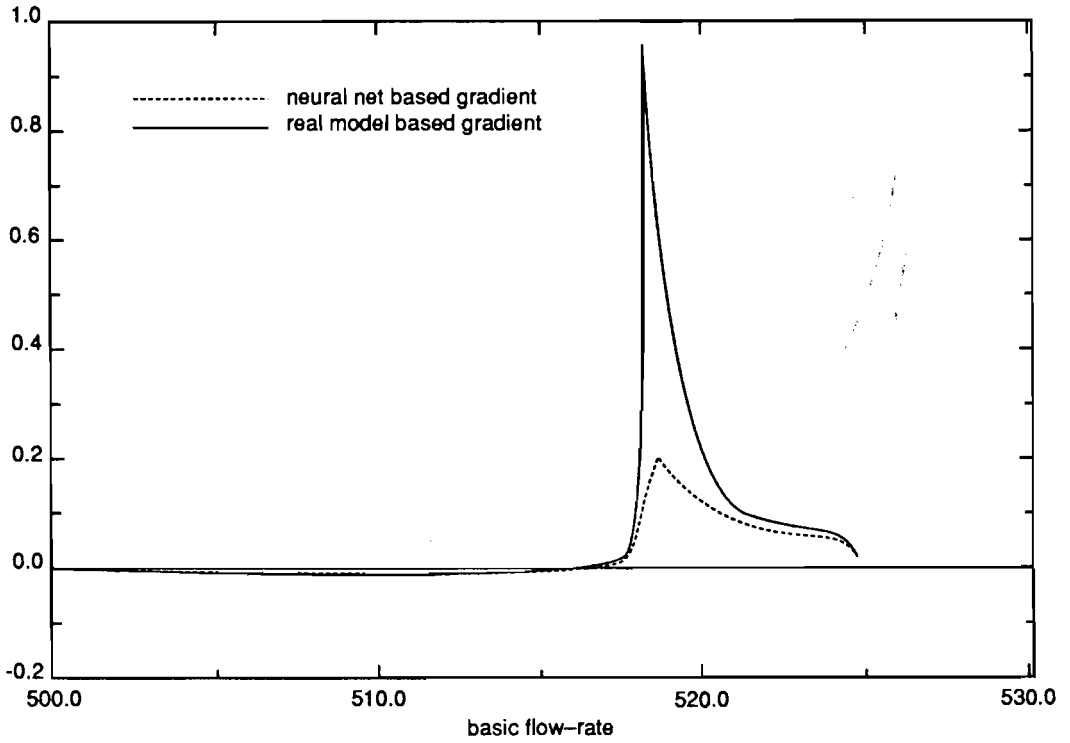


Figure 10.10 First gradient of the objective function.

information, on which it hasn't been trained. (It is possible, in principle, to augment the error function $E(t)$ by adding the length of the difference between the *gradient* vectors of the actual model and the ANN. The adaptation is straightforward and inexpensive, using second derivative techniques analogous to those used in "GDHP" [28]; however, actual gradient vectors are not generally available for real plants.) Perhaps such derivative information would be more accurate if narrower training regions were used, or if a larger network were used (with a larger training set). Other traditional chemical engineering processes should not be so demanding in terms of their nonlinear behavior compared to the pH system studied in this section.

10.4. TOWARDS A MORE ROBUST TRAINING OR IDENTIFICATION PROCEDURE: EMPIRICAL RESULTS, ALTERNATIVE MODELS, AND THEORY

The previous section has shown how the quality of system identification with an ANN can have a strong effect on the resulting quality of control. Chapters 3 and 13 argue that the quality of system identification is crucial to the capabilities of neurocontrol in general, even when other designs are

used. Most of the failures of neurocontrol in the past year or two can be traced back to inadequate system identification. This section will discuss alternative ways of adapting ANNs for the purpose of prediction, which appear to work far better than the conventional methods described in section 10.3.

The problem of system identification is not unique to neural networks. In the control theory community as well, a variety of techniques have been tried. Linear system identification has made great progress, and many generic linear models, such as transfer-function models and convolution models, are available and widely used. For example, transfer function models have been used as fundamental tools for the design and analysis of control systems. Convolution models, such as a finite impulse response model (FIR) or a step response model, have played an essential role in the dynamic matrix control algorithm (DMC). However, linear models can only be used under limited conditions and special assumptions, because most physical processes usually behave nonlinearly.

This section will begin by discussing new empirical results, obtained with an alternative training procedure in the *nonlinear* case [29]. The ANN that we adapted was a very simple static network, as in section 10.3. First we will describe how we adapted the network. Next, we will describe the real-world data and results, first for the wastewater treatment plant and then for the oil refinery. Next, in section 10.4.5, we will compare our simple ANN model and design against similar designs—such as ARMA models and their ANN generalizations—which could lead to better performance in more demanding applications. Finally, in section 10.4.6, we will describe the deeper statistical issues that underlie our training procedure, along with the *nonneural* tests of similar methods and their extensions. We will describe some alternative training methods that are likely to be more powerful than what we tested here. The issues here are extremely subtle, and there is clearly room for further research to do even better. Section 10.8 will provide some of the equations needed to implement these models and methods.

The reader should be warned that the methods described in the first part of this section will work very poorly in some situations. That is the reason why section 10.4.6 is important for some applications and why more research is needed. In some applications, however, the simple methods of section 10.4.1 do lead to an orders-of-magnitude reduction in prediction errors.

Even before we discuss our empirical results, we should say a few words about our use of the word "robust." In control theory, "robust control" typically refers to the design of linear controllers that are "robust" in the sense that they achieve stability (if not optimality) in control even as the parameters of the unknown linear plant vary over a wide interval. This is merely one special case of "robustness"—the effort to build systems that work well in practice *despite* our lack of perfect information about the plant to be controlled *or* predicted. The phrase "robust estimation" was used long ago in the field of statistics [30], to refer to identification procedures that tend to yield good predictions despite our lack of knowledge about the equations governing the real world. The effort described here was motivated by earlier work on robust estimation [3]. Again, the theory involved is quite subtle and will be described in more detail in section 10.4.6.

10.4.1. Networks and Procedures Used in System Identification in Our Tests

10.4.1.1. Structure of the ANNs In all these tests, we used a simple MLP, as described in equations 9 through 12. However, the vectors $Y(t)$ to be predicted had more components here. In the three wastewater treatment tests, we tried to predict the concentrations of three ions coming out of

the various treatment subsystems: NH_4^+, NO_3^{2-}, and PO_4^{3-}; thus, $Y(t)$ had three components, corresponding to these three concentrations, in each of these tests. In the oil refinery test, we tried to predict two production variables that we cannot describe because of the proprietary nature of the data; thus $Y(t)$ had two components. In all of these tests, the *inputs* to each network included data on other variables (u) from earlier times. The inputs *also* included $Y(t-1)$ in all cases. In two of the four tests, we also supplied $Y(t-2)$ and $Y(t-3)$ as inputs to the network. The approach we used here would *not* be useful in cases where $Y(t-1)$ is unimportant as an input in predicting $Y(t)$; however, cases like that are relatively rare in control applications.

10.4.1.2. Three Alternative Approaches to System Identification This section will only describe the two approaches tested here, and a related approach introduced for the sake of explanation. See section 10.4.6 for other approaches.

In all four tests, we used an ANN to predict $Y(t)$ as a function of earlier data. We may call this vector-valued function "F," and we may write our model as:

$$\hat{Y}(t) = F(Y(t-1), Y(t-2), Y(t-3), u(t-1), u(t-2), u(t-3), W), \tag{21}$$

where F is the function that represents the neural network, and W is the set of weights we are trying to estimate/identify/adapt.

The most common way to estimate/identify/adapt a function of this kind is to pick W so as to minimize:

$$E = \tfrac{1}{2} \sum_{ij} (Y_i(t) - \hat{Y}_i(t))^2. \tag{22}$$

Many alternative error functions (such as the 1.5 power of error) have been used for systems that do not have Gaussian noise distributions [30]; however, that aspect of robust estimation is not our major concern here. (Also, it is trivial to adapt our approach to such alternatives.) In our case, we will always minimize error as defined in equation 22, but we will minimize three different measures of error based on three different ways of interpreting equation 22. The meaning of equation 22 is not well specified until we specify how to calculate "$\hat{Y}_i(t)$" as a function of W and of our data. The most common procedure, based on maximum likelihood theory and a simple model of noise, is to substitute equation 21 into equation 22 and minimize the following expression as a function of W:

$$E_1 = \tfrac{1}{2} \sum_{ij} (Y_i(t) - F_i(Y(t-1), Y(t-2), Y(t-3), u(t-1), u(t-2), u(t-3), W))^2 \tag{23}$$

where $Y(t-\ldots)$ refers to actual observed data for Y, and where F_i refers to the ith component of the function F. This is the procedure we used in section 10.3, and it is also the procedure used in the bulk of ANN research.

An alternative approach is to build up prediction for $Y(t)$ from *predicted* values in earlier time periods. Thus, we may calculate a prediction:

$$\hat{Y}(t) = F(\hat{Y}(t-1), \hat{Y}(t-2), \hat{Y}(t-3), u(t-1), u(t-2), u(t-3), W) \tag{24}$$

and therefore try to minimize the following function of W, treating the other arguments of the function as constants:

$$E_2 = \tfrac{1}{2} \sum_{t,i} (\hat{Y}_i(t) - F_i(\hat{\mathbf{Y}}(t-1), \hat{\mathbf{Y}}(t-2), \hat{\mathbf{Y}}(t-3), \mathbf{u}(t-1), \mathbf{u}(t-2), \mathbf{u}(t-3), W))^2. \tag{25}$$

Approaches of this sort have been used quite often in the linear case in control theory, particularly by Ljung and by Narendra [31–33]. Usually they are called parallel identification methods.

Notice that equations 23 and 25 are *both* defined, at time t, as functions of a small number of inputs available at time t (and earlier). Therefore, both equations would allow us to adapt an ANN on an observation-by-observation basis, as in section 10.3. Both are suitable for *real-time* system identification. In these tests, however, we studied the problem of *off-line* system identification.

In off-line system identification, it is possible to do something very similar to equation 25, while *also* accounting for the influence of the weights W in *changing* the values of $\hat{\mathbf{Y}}(t-1)$, $\hat{\mathbf{Y}}(t-2)$, and $\hat{\mathbf{Y}}(t-3)$, as they are input to the neural net. To explain this idea in formal terms, we need to consider how equation 24—chained back from time t to time 4 (where time $t = 1$ is defined as the earliest data point)—leads to the functional relationships:

$$\begin{aligned}
\hat{\mathbf{Y}}(t) &= \mathbf{F}(\hat{\mathbf{Y}}(t-1), \hat{\mathbf{Y}}(t-2), \hat{\mathbf{Y}}(t-3), \mathbf{u}(t-1), \ldots \mathbf{u}(t-3), W) \qquad (26)\\
&= \mathbf{F}(\mathbf{F}(\hat{\mathbf{Y}}(t-2), \hat{\mathbf{Y}}(t-3), \hat{\mathbf{Y}}(t-4), \mathbf{u}(t-2), \ldots \mathbf{u}(t-4), W), \hat{\mathbf{Y}}(t-2), \hat{\mathbf{Y}}(t-3), \mathbf{u}(t-1) \ldots W)\\
&= \mathbf{F}(\mathbf{F}(\mathbf{F}(\hat{\mathbf{Y}}(t-3) \ldots W), \hat{\mathbf{Y}}(t-3), \hat{\mathbf{Y}}(t-4), \mathbf{u}(t-2) \ldots W), \mathbf{F}(\hat{\mathbf{Y}}(t-3), \ldots), \hat{\mathbf{Y}}(t-3), \mathbf{u}(t-1) \ldots W)\\
&= \mathbf{G}(\mathbf{Y}(3), \mathbf{Y}(2), \mathbf{Y}(1), \mathbf{u}(t-1) \ldots \mathbf{u}(1), W),
\end{aligned}$$

for a function $\mathbf{G}$ which is fairly complex. In other words, equation 24 implicitly makes $\hat{\mathbf{Y}}(t)$ a function of $\mathbf{Y}(3), \ldots, W$. In our tests, we picked W so as to minimize the following measure of error as a function of W:

$$E_3 = \tfrac{1}{2} \sum_{i,j} (Y_i(t), - G_i(\mathbf{Y}(3), \mathbf{Y}(2), \mathbf{Y}(1), \mathbf{u}(t-1) \ldots \mathbf{u}(1), W))^2. \tag{27}$$

In practice, the idea is simply to *chain* the network forward from initial data, as shown in Figure 10.11 (drawn for a different application), and to minimize the total error across the *entire time trajectory*.

In other words, the classic real-time version of parallel identification treats $\hat{\mathbf{Y}}(t-1)$ through $\hat{\mathbf{Y}}(t-3)$ as *constants*, when adjusting W so as to minimize error at time t. The off-line version attempts to account for the impact of the weights in *changing* what these inputs would have been. This approach can be applied in a real-time system as well (section 10.4.6), but the tests reported here were all done off-line. The approach can be applied to any forecasting model—not just ANNs—but the minimization problems become much easier when the backpropagation algorithm is used.

The idea of adapting ANNs so as to minimize equation 27 was first proposed by Werbos [34,35], as an extension of his earlier work on the general *nonneural* case [3,36,37]. The ANN version has also been used by Kawato on data from a simulated robot arm, as part of his "cascade" control design

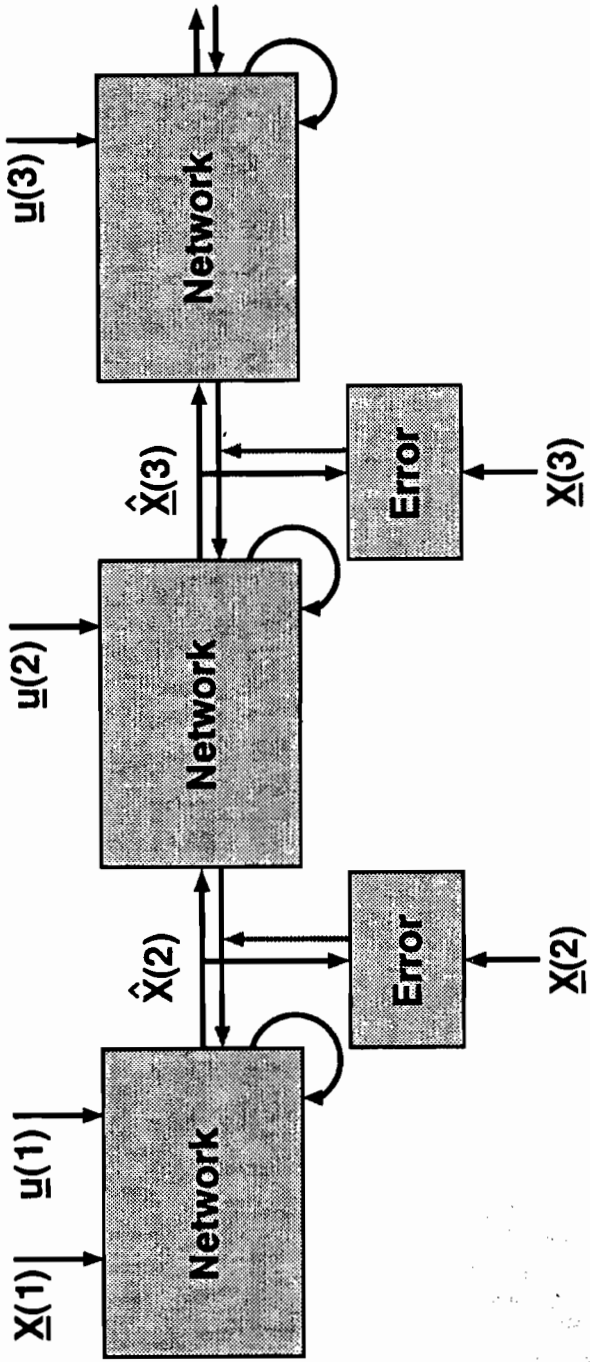


Figure 10.11 The pure robust method.

[38]. Werbos has called this approach "the pure robust method." In some situations, it is better to use a more complex method, called the compromise method, to be discussed in section 10.4.6. The pure robust method can be rationalized as an alternative maximum likelihood method, based on what control theorists have called an output errors assumption [31–33]; however, section 10.4.6 will argue that this interpretation understates the significance of the method.

In the tests reported here, we compared the conventional maximum likelihood approach (equation 23) against the pure robust method (equation 27) to see which led to better predictions from the ANN. The pure robust method led to multiperiod prediction errors that were smaller by orders of magnitude.

10.4.1.3. Overview of the Algorithm It is possible, in theory, to minimize equation 27 by the usual procedure of "backpropagation through time" (BTT). More precisely, we can pick initial values of the weights W by some sort of random number generator and iterate through the following steps until the weights settle down:

1. Starting from actual data on Y in times $t = 1$ through $t = 3$, and actual data on $\boldsymbol{u}$ at all times, use equation 24 to calculate $\hat{Y}(4)$; then use it to calculate $\hat{Y}(5)$, and so on up to the end of the training data.
2. Calculate the sum of square error, E, across all time (based on equation 22).
3. Update all the weights, W_i, by:

$$\text{new } W_i = \text{old } W_i - \textit{learning\_rate} * \frac{\partial E}{\partial W_i},$$

for some small *learning_rate* determined in ad hoc fashion. The calculation of the derivatives of E with respect to the W_i can be done all at once, in one quick sweep through the data, using the backpropagation algorithm.

Unfortunately, the errors E depend on the weights W in a highly volatile nonlinear way. There is a kind of Murphy's law at work here: Highly accurate statistical estimation implies very large second derivatives (because of how one calculates estimation errors in statistics); large second derivatives tend to imply nonsmooth surfaces with steep regions near the optimum; that, in turn, makes it harder to find the minimum of error. In earlier work, Werbos has generally used the following procedure to overcome these problems:

1. Do *not* initialize the weights at random. Instead, initialize them to their maximum likelihood values.
2. Next, adapt the weights using some form of the compromise method, which is similar to the pure robust method, but not so extreme.
3. Then use *those* weights as the initial weights with the pure robust method. (If necessary, repeat step two several times, using stiffer and stiffer forms of compromise.)

This procedure is not guaranteed to find a global minimum. However, it *is* guaranteed to do *better*, in some sense, than the initial weights—the maximum likelihood weights, which are the major alternative we had available to us. In addition, this procedure will work much better if one uses

adaptive learning rates, as given in Chapters 3 and 13, or a sophisticated numerical method like one of those discussed by Shanno [24].

For these tests, it was easier and simpler to alternate between a simple version of BTT (as given at the start of this section) and a random search procedure in order to avoid a local minimum. Convergence was still essentially an overnight affair, but this was easier for us than spending weeks to upgrade the computer program. In addition, we used an iterative procedure to decide on the number of hidden units in each network. The remainder of this section will describe the remaining elements of the algorithm in more detail.

10.4.1.4. Calculation of the Derivatives of Error The derivatives of E_3, as defined in equation 27, can be calculated, in principle, by using the *conventional* chain rule for partial derivatives. However, this calculation is very tedious and very subtle. In addition, it is not a "linear" kind of calculation, which can easily be automated. Even the best engineers make errors very easily when they try to calculate partial derivatives in an ad hoc way, through very complex systems.

Werbos [3] developed an alternative chain rule in 1974 for use with ordered dynamical systems, which is more "linear" (i.e., straightforward) and was proven to yield the correct derivatives. Section 10.6 describes the use of this chain rule in more detail.

In our application, it was easiest to write out the forwards equations which were used to calculate error, and then to apply the chain rule for ordered derivatives *directly* to our system, in order to work out the derivative equations. Section 10.8 will provide a more general version of these equations; here, we will report exactly on the version used in these tests [29], developed by Su, for the sake of the intuition that it may provide.

The forwards equations of the MLP were represented as:

$$\nu^l(t) = W^l x^{l-1}(t) \tag{28}$$

$$x^l(t) = s_l(\nu^l(t)),\ l = 1,2,...,L, \tag{29}$$

where L denotes the output layer (there are $L+1$ layers in all), and $\nu^l(\,)$ and $x^l(\,)$ are the voltage vector and the output vector, respectively, from the lth layer. The various vectors, matrices, and functions are defined as:

$$W^l \triangleq \begin{vmatrix} 1 & 0 & \dots & 0 \\ W^l_{1,0} & W^l_{1,1} & \dots & W^l_{1,n^{l-1}} \\ W^l_{2,0} & W^l_{2,1} & \dots & W^l_{2,n^{l-1}} \\ \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \dots & \cdot \\ W^l_{n^l,0} & W^l_{n^l,1} & \dots & W^l_{n^l,n^{l-1}} \end{vmatrix} \tag{30}$$

$$s_l(\nu) \triangleq (1, s(\nu_1), ..., s(\nu_{n^l}))^T \tag{31}$$

$$v^\iota(t) \triangleq (1, v_1(t), ..., v_{n^\iota}(t))^T \tag{32}$$

$$x^\iota(t) \triangleq (1, x_1(t), ..., x_{n^\iota}(t))^T . \tag{33}$$

Here, $W_{j,0}^\iota$ are bias weights and $s(\,)$ is the sigmoid function defined in equations 10 and 12. In addition, an input neuron can be represented as a neuron with a linear activation function; in other words, $x^0(t) = v^0(t)$. These equations, combined with equations 22 and 24, complete our description of how we calculated total error as a function of the weights.

Starting from those system equations, the chain rule for ordered derivatives leads very simply and directly to the following relationships:

$$\frac{\partial^+ E}{\partial W_{ji}^\iota} = \sum_{t=1}^{T} \frac{\partial^+ E}{\partial v_j^\iota(t)} \cdot \frac{\partial' v_j^\iota(T)}{\partial W_{ji}^\iota} \tag{34}$$

$$\frac{\partial^+ E}{\partial v_j^\iota(t)} = \frac{\partial^+ E}{\partial x_j^\iota(t)} \cdot \frac{\partial' s_j^\iota(t)}{\partial v_j^\iota(t)} \tag{35}$$

$$\frac{\partial^+ E}{\partial x_j^\iota(t)} = \frac{\partial' E}{\partial x_j^\iota(t)} + \sum_{k=1}^{n^{\iota+1}} \frac{\partial^+ E}{\partial v_k^{\iota+1}(t)} \cdot \frac{\partial' v_k^{\iota+1}(t)}{\partial x_j^\iota(t)} \tag{36}$$

and $\partial'/\partial x_j^\iota(t) = 0$ for all $\iota \neq L$, where ∂^+ indicates ordered derivatives and ∂' indicates conventional partial derivatives. These conventional partial derivatives are calculated by differentiating the functions for E, v, and x exactly as they are written in equations 22, 28, and 29 without any substitutions. Let us define:

$$F\_v_j^\iota(t) = \delta_j^\iota(t) = \frac{\partial^+ E}{\partial v_j^\iota(t)} .$$

With this notation, equation 34 looks very similar to the original delta rule of Widrow and Hoff [39]:

$$\frac{\partial^+ E}{\partial W_{ji}^\iota} = \sum_{t=1}^{T} F\_v_j^\iota(t) x_i^{\iota-1}(t) = \sum_{t=1}^{T} \delta_j^\iota(t) x_i^{\iota-1}(t) . \tag{37}$$

In other words, the change of the weights W^ι that connect the $(\iota - 1)$th layer to the ι^{th} layer is determined by the product of the $(\iota - 1)^{th}$ layer's output and the ι^{th} layer's delta. Thus, the key to updating weights in the training phase is to calculate the deltas, the F_v terms, the derivative feedback to the voltage variables. This calculation can be carried out as follows:

$$F\_v_j^\iota(t) = \frac{\partial^+ E}{\partial x_j^\iota(t)} \; s_j^{\iota\prime}(t) \tag{38}$$

$$\frac{\partial^+ E}{\partial x_j^\iota(t)} = \frac{\partial' E}{\partial x_j^\iota(t)} + \sum_{k=1}^{n^{\iota+1}} F\_v_j^{\iota+1}(t) W_{kj}^{\iota+1} . \tag{39}$$

Equation 39 contributes to propagating information about error backwards from the $(\iota + 1)^{th}$ to the ι^{th} layer through the deltas recursively. Note that:

$$s_j'^{\iota}(t+1) \triangleq \frac{\partial s_j^\iota(t)}{\partial v_j^\iota(t)} = x_j^\iota(t)(1 - x_j^\iota(t)) , \tag{40}$$

for layers ι where the sigmoid function is used in equation 29; however, for linear neurons (e.g., input neurons), this s' term will simply equal one. Nevertheless, equation 39 has a slightly different form, depending on whether we calculate error in the traditional way (equation 23) or in the pure robust way (equation 27).

Case 1: When we calculate error in the traditional way, equation 39 becomes exactly the same as in basic backpropagation or the "generalized delta rule" [23], i.e.:

$$F\_v_j^L(t) = e_j(t) s_j'^L(t) \tag{41}$$

$$F\_v_j^h(t) = \left(\sum_{k=1}^{n^{h+1}} F\_v_k^{h+1}(t) W_{kj}^{h+1} \right) s_j'^h(t) \tag{42}$$

where $e_j(t)$ is defined as $x_j^L(t) - Y_j(t)$, and $h \neq L$ indicates hidden layers.

Case 2: When the ANN output $x^L(t)$ replaces $Y(t)$ as the input to the network, then changes of weights will affect $x^L(t+1)$ and thus affect $x^L(t+2)$ all the way up to $x^L(t)$. Therefore, the second term of equation 39 has to account for this chaining from time $t = 0$ to $t = T$. As mentioned earlier, the input layer at time $t + 1$ can be considered as a part of the $(L + 1)^{th}$ layer of the network at time t (see Figure 10.12). When calculating the delta of the output layer at time t, one has to calculate the delta for the input neurons at times $t + 1$ up to $t + k$, all of which are directly connected to the corresponding output neurons at time t. (Again, in our tests, we used a k—a basic time lag—of 1 in some tests, and 3 in others.) It is easy to show that:

$$F\_v_j^0(t) = \sum_{k=1}^{n^1} F\_v_k^1(t) W_{kj}^1 \tag{43}$$

since the input neurons are linear. Therefore, the delta for an output neuron becomes:

$$F\_v_j^L(t) = \left(e_j(t) + \sum_{\tau=1}^{k} F\_v_*^0(t+\tau) \right) s_j'^L(t). \tag{44}$$

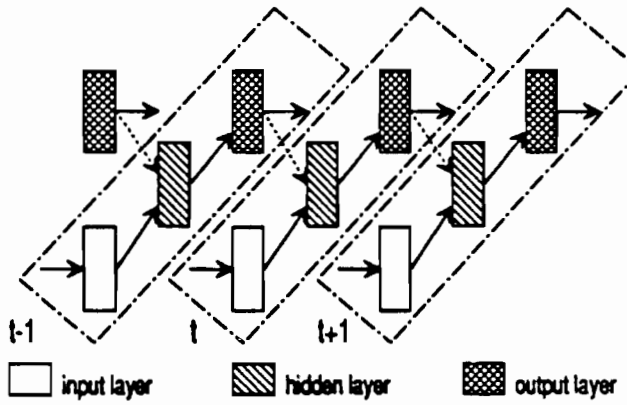


Figure 10.12 Another view of the time-lag recurrent network.

Here, * indicates the delta of an appropriate neuron to which the output neuron is connected. In equation 44, the delta propagates the required information all the way from $t = T$ back to the current time t; therefore, the delta rule for a time-lagged recurrent system is referred to as the backpropagation through time (BTT) algorithm [22,40]. Likewise, we can easily define a similar rule for training networks where equation 28 is generalized so that v^l depends on $x(t-1)$ in addition to $x(t)$, as in FIR neural networks (see section 10.4.5 and section 10.8).

10.4.1.5. Random Search Algorithm Random search methods have been widely used in classical optimization problems. It has been proven that certain random search methods can always reach the global minimum of a nonlinear function. Baba [41] has pointed out that the random search algorithm by Solis and Wets [42] can lead to a global minimum for training a multilayer perceptron. Morris and his co-workers [43] have also utilized a random search algorithm (Chemotaxis algorithm) for neural network training [44]. In these tests, Solis' and Wets' algorithm was used for robust training whenever a convergence problem was encountered. This algorithm, according to its authors, ensures eventual convergence to a global minimum of the objective function. (In actuality, of course, there is no way to *guarantee* global convergence in less than exponential time; see section 10.4.6 for related issues.) The procedure was as follows:

1. Select an initial estimate for W^k and let $b^0 = 0$.
2. Generate a Gaussian random vector e^k with a predefined variance and a mean of b^k.
3. If $E(W^k + e^k) < E(W^k)$, set $W^{k+1} = W^k + e^k$ and $b^{k+1} = 0.2b^k + 0.4e^k$.
4. If $E(Wk + ek)$ ô $E(Wk) > E(Wk - e^k)$, set $W^{k+1} = W^k - e^k$ and $b^{k+1} = b^k - 0.4e^k$.
5. Otherwise, let $W^{k+1} = W^k$ and $b^{k+1} = 0.5b^k$.
6. Repeat step two until convergence.

10.4.1.6. Determining the Number of Hidden Units Since sufficient hidden neurons are essential for a network to approximate a nonlinear function, the determination of the number of hidden neurons is crucial in the training phase. Many researchers have considered this issue, although no general and efficient method is available. In our tests, we simply used a cross-validation process as a procedure for determining the number of hidden units. First, the number of hidden neurons for the maximum likelihood version of the network is determined. Then, for ease of comparison, the number of hidden units for the robust version is set equal to that of the maximum likelihood version.

The cross-validation procedure was as follows:

1. Split the data set into a training set and a testing set.
2. Start with a small number of hidden units.
3. Initialize all the weights to zero as discussed below.
4. Apply the generalized delta rule to update the weights.
5. Validate the updated model with the testing set.
6. Repeat the training process until the prediction error of the testing set reaches a minimum.
7. Add an additional hidden neuron to the network.
8. Repeat step three until adding additional hidden neurons increases the prediction error on the testing set.

In order to have a common starting point for different-size networks, all the weights are initialized to zero when determining the number of hidden neurons. A network with all weights equal to zero, regardless of the number of hidden neurons, will have the property that the prediction errors for both training and testing sets are proportional to the sampling variance of the whole data set. However, for a network with all weights equal to zero, the final results for the weight matrix will result in all the hidden neurons being identical to each other. To avoid this problem, different learning rates are assigned to the weights connected with different hidden neurons. In this work, the learning rate is multiplied by a linearly decreasing factor varying with the hidden unit so that the learning rate is actually a vector, equal to a global learning rate multiplied by $(1,1,\ldots,2/n^l,1/n^l)$. Note that the bias weight has the same learning rate as the first hidden neuron. From an optimization point of view, one can consider that the gradient is being multiplied by the matrix $\text{diag}[1,1,\ldots,2/n^l,1/n^l]$, which is positive-definite and invertible. It is easy to prove that an optimization algorithm using such a transformed gradient can lead to a minimum in the error.

10.4.2. Example Number One: A Wastewater Treatment System

10.4.2.1. Description of the System and the Network The data for this study were provided by Pedersen, Kummel, and Soeberg [45], to whom we owe many thanks.

A wastewater treatment plant is a very complex process, especially when biological removal of nitrogen, phosphorous, and organic material (N, P, and C) is implemented, as in the research of Pedersen et al. [45]. The pilot plant, as sketched in Figure 10.13, is fed with clarified municipal wastewater, in which the large particles, sand, and oils have been removed. The process is carried out in four vessels, in which the biological removal of phosphorous, nitrogen, and organic material takes place. The throughput of the pilot plant is on the order of 60 liters per hour. The approximate

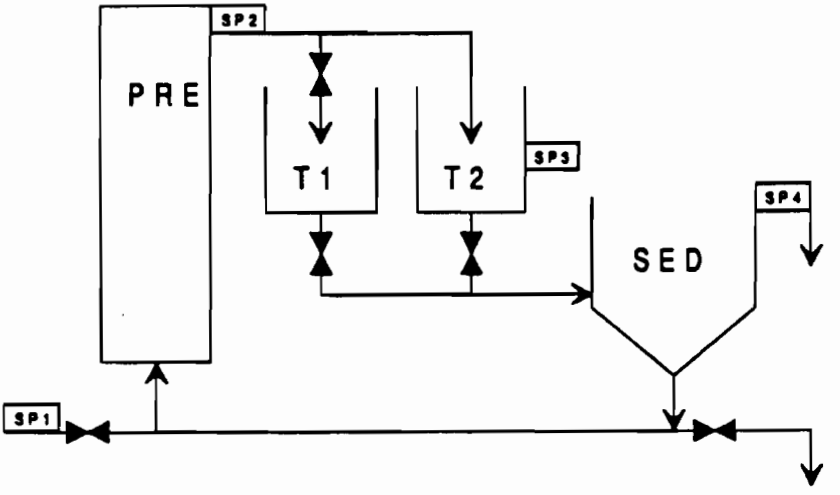


Figure 10.13 The process flow diagram of the wastewater treatment plant.

holding volumes of the pretreatment column (PRE), the two aeration tanks (T1 and T2), and the sedimentation tank (SED) are 200, 800, 800, and 1,100 liters, respectively.

The feed to the pretreatment column is a mixture of raw wastewater and return sludge. The pretreatment column is maintained anaerobic, and the main biological process is carried out by the phosphorus accumulating bacteria. The two aeration tanks T1 and T2 are operated in an alternating manner that is scheduled prior to the operation. The nitrification and denitrification processes take place in these two tanks alternatively. In the sedimentation vessel, the heavier sludge falls out of suspension. From the top of the vessel, the process effluent is removed from the system. The sludge is returned via the sludge return pump and is mixed with the fluid entering the pretreatment column. The sludge concentration is held approximately constant through intermittent removal of sludge from the system at the point where it exits the sedimentation vessel. Samples are taken from four different locations as indicated in Figure 10.13, in which SP_1 through SP_4 measure the concentrations of NH_4^+, NO_3^{2-} and PO_4^{3-} in the incoming stream, the pretreatment column, the aeration tanks, and the sedimentation vessel, respectively. The oxygen consumption rates at the aeration tanks are also recorded. A time series of four days' data is used to model the pilot plant. The data set is obtained by sampling the system every seven minutes. This sampling results in nine hundred data points in total. All the variables are linearly scaled between 0.1 to 0.9. The ranges of all the variables are given in Table 10.2. The data is split into a training set (the first six hundred samples) and a testing set (the remaining three hundred samples). The outputs of interest are the concentration measurements for the three processing units (nine variables). The inputs are the concentrations of the incoming stream and the oxygen consumption rates (five variables).

One can construct a network to model the whole plant as a 5 by 9 multi-input–multioutput (MIMO) system. Alternatively, in this study we used three smaller networks to model the three processing

Table 10.2 The range for each of the measurements for the wastewater plant

unit: mg/l	Inlet	PRE	T12	SED
$[NH_4^+]$	.0366 49.51	4.956 28.934	.0366 11.768	.0061 1.147
$[NO_3^{2-}]$	0.000 11.58	.000 .824	0.0214 16.290	.4364 12.50
$[PO_4^{3-}]$	.1953 8.287	11.130 48.276	0.5066 21.658	.6195 9.338

Table 10.3 The number of neurons for each layer of the neural network for the wastewater plant and the catalytic reformer

	PRE	T12	SED	REF
input	6	8	18	21
hidden	5	5	6	6
output	3	3	3	2

units individually. The three processing units are the pretreatment column (PRE), the aeration tanks (T12), and the sedimentation vessel. The pretreatment column is a 3 by 3 system, the aeration tanks, 5 by 3, and the sedimentation vessel, 3 by 3. A first-order dynamic model (k = 1) is chosen for PRE and T12, and a third order for SED. The order of the dynamic model determines the number of input neurons needed for the networks. The size of the network for each subsystem is summarized in Table 10.3.

10.4.2.2. Results on the Wastewater Plant After the ANN is trained by the maximum likelihood procedure, the one-step-ahead prediction by the ANN for each subsystem appears to be exceptionally good. These results for the three subsystems (PRE, T12, and SED) are shown in Figures 10.14, 10.15, and 10.16, respectively. In principle, one can use these models for nonlinear model predictive control (MPC), as described in section 10.3. In MPC, the model usually predicts many time-steps ahead into the future. Therefore, the feedforward network, which predicts one step ahead, has to be iterated; in other words, the network has to be chained to itself to go as far as needed into the future. For example, if a predicted output in the future, like $\hat{Y}(t+k)$, is needed at any time t, then the predicted output $\hat{Y}(t+1)$ has to replace the actual output $Y(t+1)$ measured from the system, and $\hat{Y}(t+2)$ must replace $Y(t+2)$ and so on, all the way up to $t+k-1$, because the actual system outputs in the future are not yet known. Therefore, the errors of importance to MPC are the multiperiod forecasting errors, the kinds of errors minimized in the pure robust method. In this situation, the gap between Y and $\hat{Y}$ can grow in a cumulative fashion over time, leading to large errors.

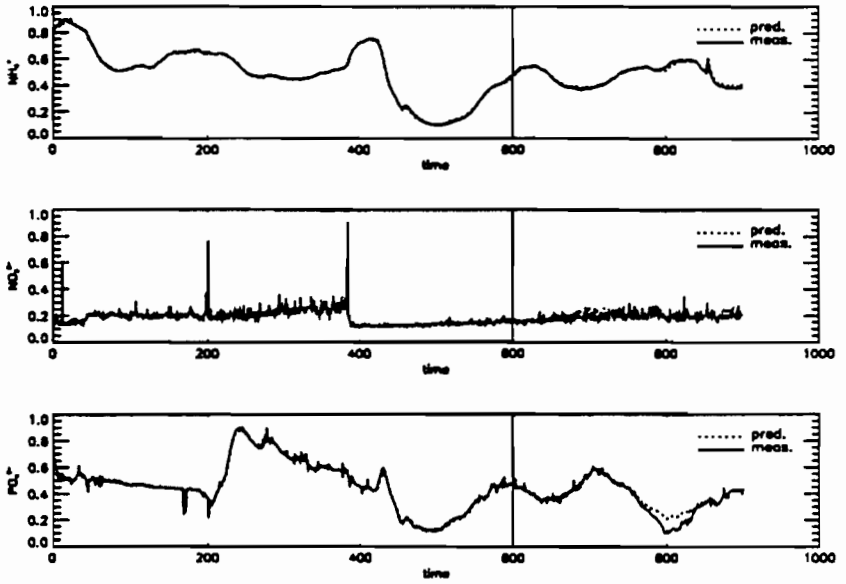


Figure 10.14 FFN: one-step ahead prediction for PRE.

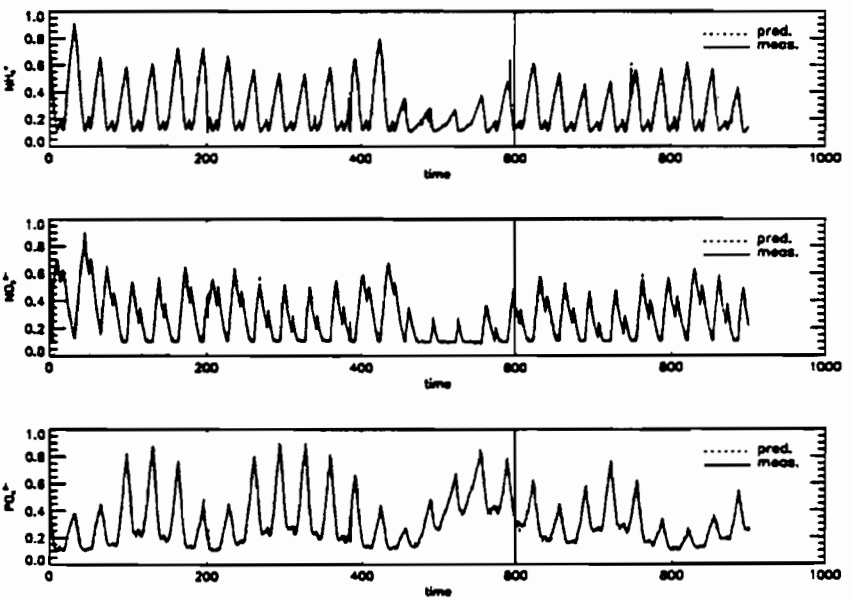


Figure 10.15 FFN: one-step ahead prediction for T12.

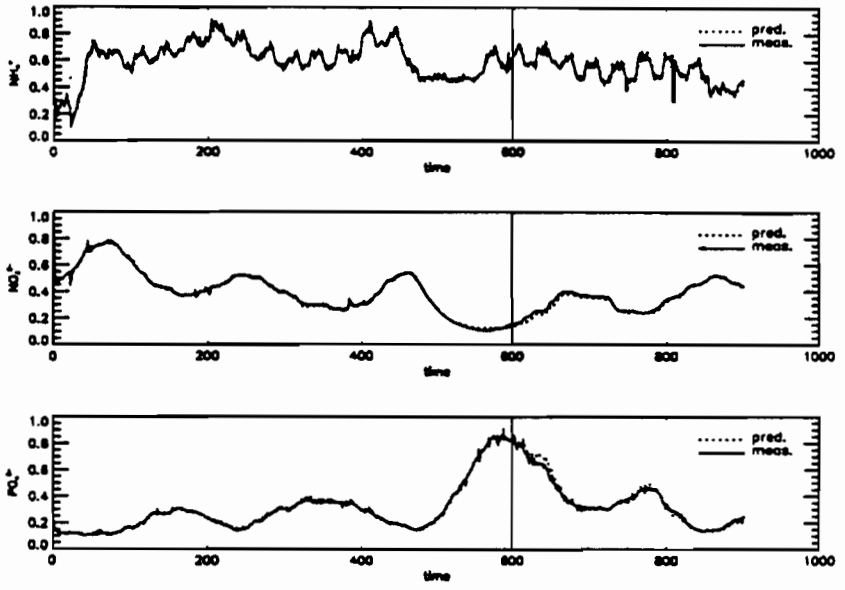


Figure 10.16 FFN: one-step ahead prediction for SED.

As an extreme example, the maximum likelihood net (MLN) is chained to predict the future system output all the way from $t = 1$ to $t = 900$, including the training set as well as the testing set. As shown in Figures 10.17 through 10.19, the MLNs do not give consistently good results for the long-term predictions on the three subsystems as they do for the one-step-ahead predictions. However, the MLN, a one-step-ahead predictor, can give a fairly good long-term prediction for the T12 subsystem. For the PRE and SED subsystems, as shown in Figures 10.17 and 10.19, the MLNs do not give reasonable long-term predictions at all. The reason for such inconsistent results from the use of MLNs is simply that they are not trained to make long-term predictions.

On the other hand, the networks adapted by the pure robust method are trained to predict the whole trajectory of the system output all the way from $t = 1$ to $t = T$ ($T = 600$). Figures 10.20 through 10.22 show the results from the pure robust networks (PRNs) in predicting from time $t = 1$ to $t = 900$ (105 hours ahead). Clearly, the PRN is significantly superior to the MLN for long-term predictions, especially for the PRE and SED subsystems. For comparison, the root mean square (RMS) of the prediction errors for each subsystem are given in Table 10.4. Note that the prediction errors are based on the scaled variables.

10.4.3. Example Number Two: A Catalytic Reforming System

10.4.3.1. Description of the System and the Network The data for this test came from Mobil Corporation, whose help is gratefully acknowledged.

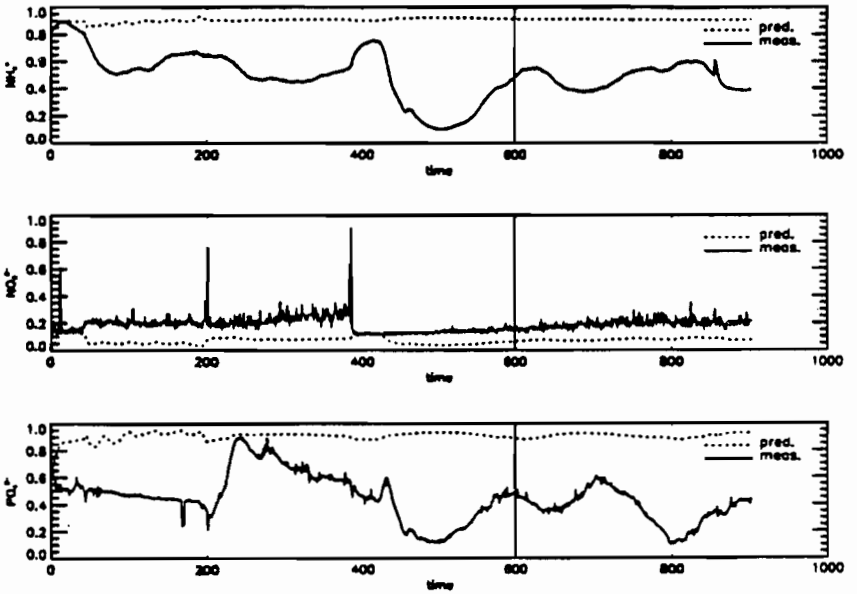


Figure 10.17 FFN: long-term prediction for T12.

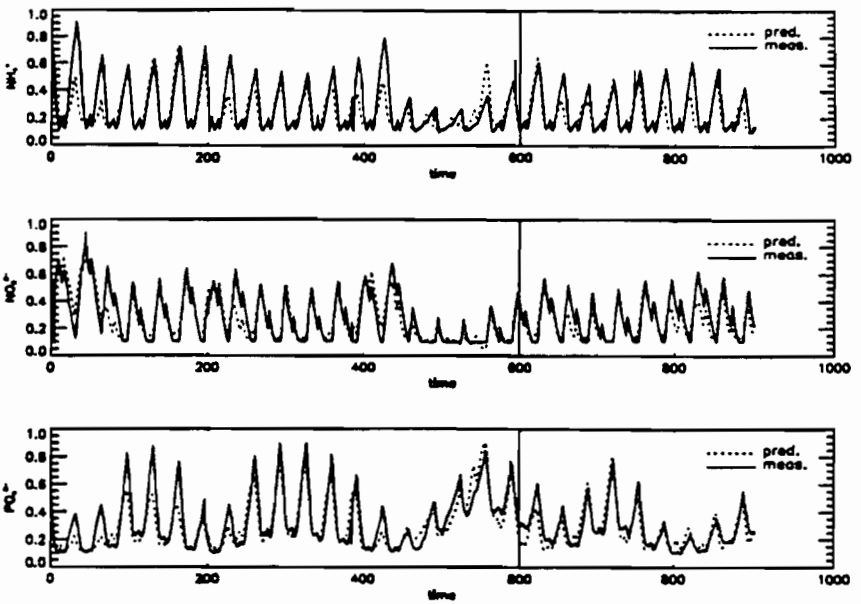


Figure 10.18 FFN: long-term prediction for T12.

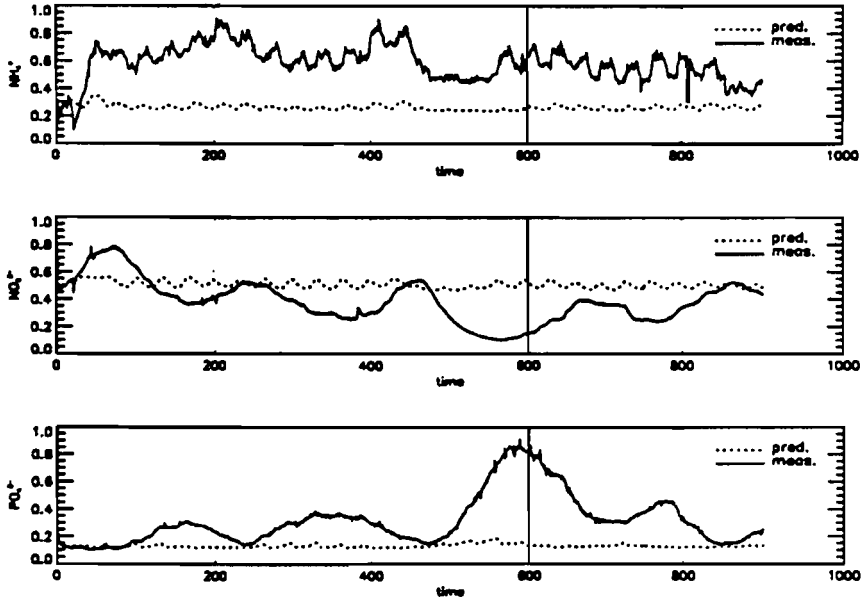


Figure 10.19 FFN: long-term prediction for SED.

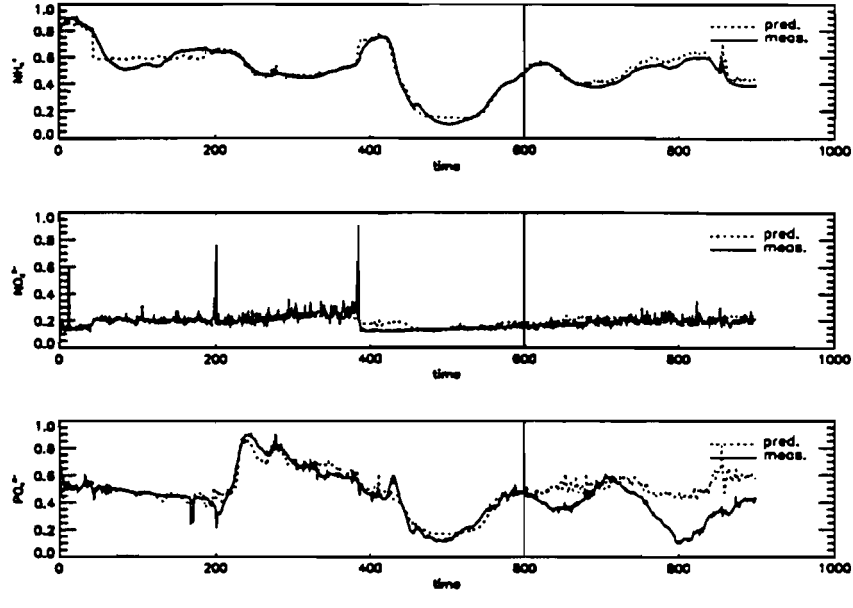


Figure 10.20 REC: long-term prediction for PRE.

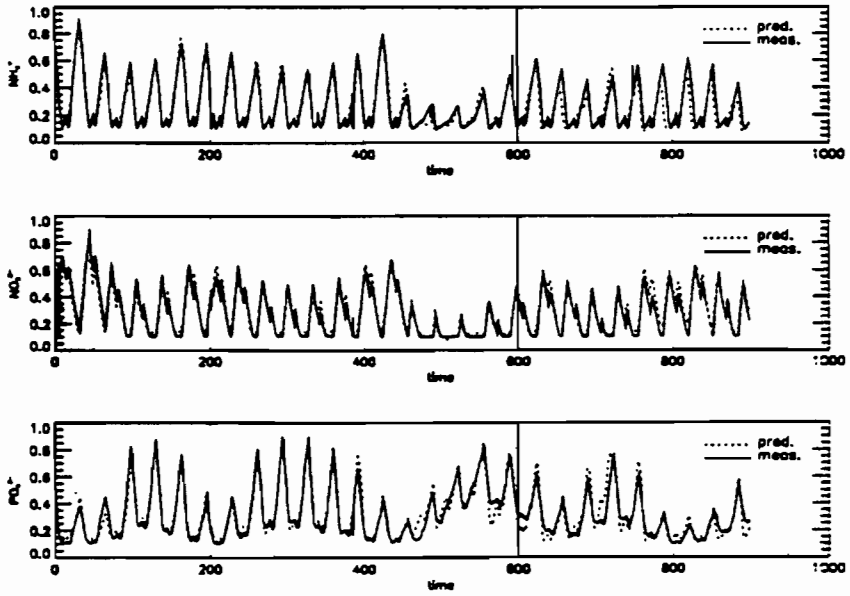


Figure 10.21 REC: long-term prediction for T12.

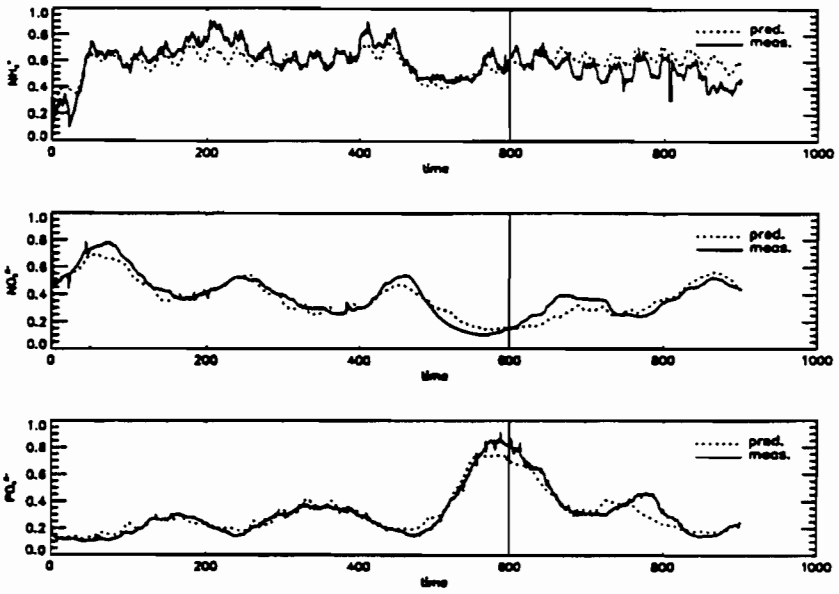


Figure 10.22 REC: long-term prediction for SED.

Since the 1970s, the catalytic reformer has become one of the key systems in oil refineries, and catalytic reformate has furnished a relatively high portion of the gasoline used in the United States [46]. In catalytic reforming, the structure of the hydrocarbon molecules is rearranged to form higher octane aromatics that can be used to increase the octane number of gasoline. The catalytic reformer carries out four major reactions: (1) dehydrogenation of naphthenes to aromatics; (2) dehydrocyclization of paraffins to aromatics; (3) isomerization; and (4) hydrocracking. Catalyst activity is reduced during operation by coke deposition and chloride loss, and it can be restored by chlorination following high temperature oxidation of the carbon.

A typical catalytic reforming process consists of a series of reactors and heaters, and a fractionator, as shown in Figure 10.23. The typical feed to a catalytic reformer is the heavy straight-run (HSR) gasoline and naphtha. Material that will deactivate the catalyst is removed prior to the process. The pretreated feed and recycle hydrogen are heated to about 900° F before entering the first reactor. In the reactors, the reactions will cause temperature drops. The effluent from a reactor has to be reheated to maintain the reaction rate before it enters the next reactor. As the flow goes down the reactors, the reaction rate decreases and the reactors become larger, and the reheat needed becomes less. Usually three reactors are enough to provide the desired degree of reaction. The effluent from the last reactor is cooled and separated into hydrogen-rich gaseous and liquid streams. The gaseous stream is recycled to the first reactor and the liquid stream is sent to the fractionator to be debutanized. The reformate is drawn out from the fractionator.

The data for the catalytic reforming system (REF) contains five inputs (denoted by u's) and two outputs (denoted by Y's). Due to a secrecy agreement, the detailed description of the measurements and the range of each variable are not given. As with the wastewater plant, the variables taken from the reformer are linearly scaled to between 0.1 and 0.9. The data contains 392 data points in total, of

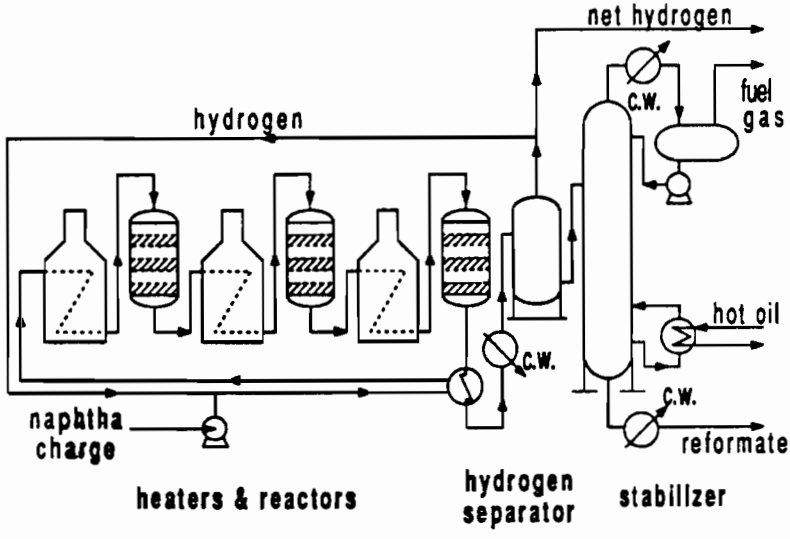


Figure 10.23 REC: long-term prediction for PRE.

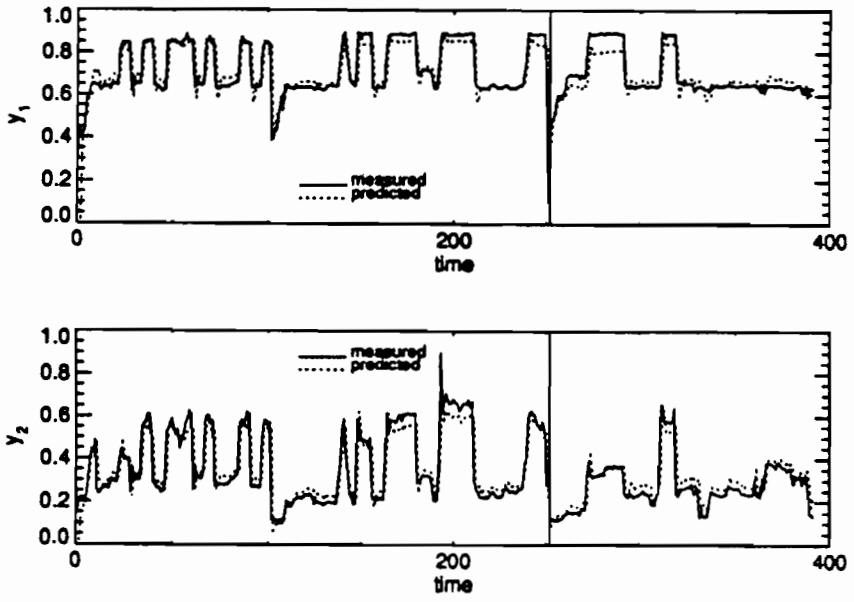


Figure 10.24 FFN: one-step ahead prediction for REF.

which the first 252 points are used for training and the remaining 140 for testing. In this case, a third-order 5 by 2 model is used. The size of the network used here is described in Table 10.3.

10.4.3.2. Results on the Catalytic Reformer In the case of one-step-ahead prediction, the network trained by the conventional maximum likelihood approach predicts the testing set of the catalytic reformer as well as it did with the wastewater system (Figure 10.24). However, it did not perform as well in tests of multiperiod predictions. In the most extreme test, the network was chained to make long-term predictions from $t = 1$ to $t = 392$ (including both training and testing sets). As in the wastewater example, the maximum likelihood network (MLN) performed much worse on these tests than it did for one-step-ahead prediction (Figure 10.25). Strictly speaking, one should not use an MLN for multiple-step predictions since it is trained to make single-step predictions. The multiple-step prediction requires iteration or chaining, and conventional maximum likelihood training does not take such chaining into account. On the other hand, the pure robust (PRN) version of the network was trained to make long-term predictions. After training, it gives much better results than the MLN for long-term predictions (Figure 10.26). For comparison, the RMS of the prediction errors for the reformer are given in Table 10.4. Notice that the prediction errors are calculated based on the scaled variables.

10.4.4. Overall Evaluation of Performance and Implications

As described earlier, networks trained by conventional maximum likelihood procedures and networks trained by the pure robust method were used to make long-term predictions. The number of prediction

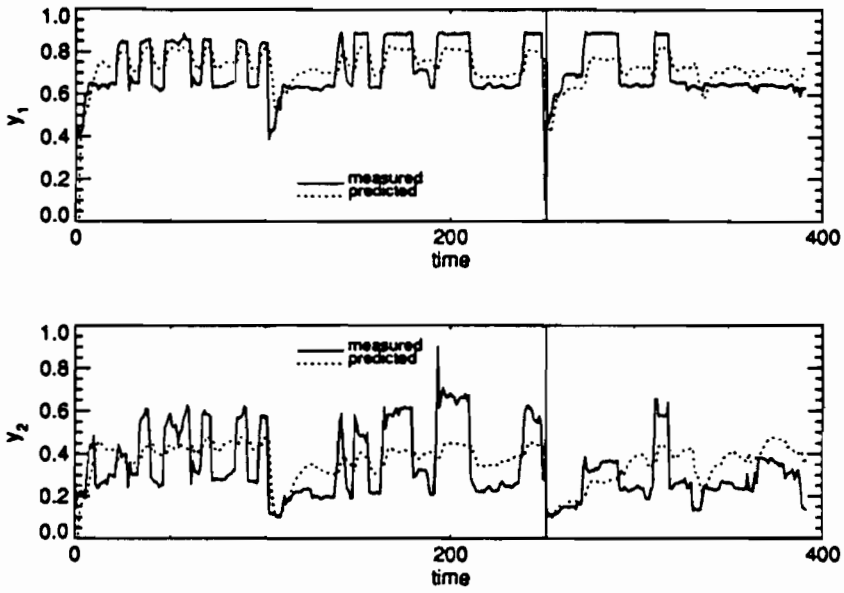


Figure 10.25 REC: long-term prediction for REF.

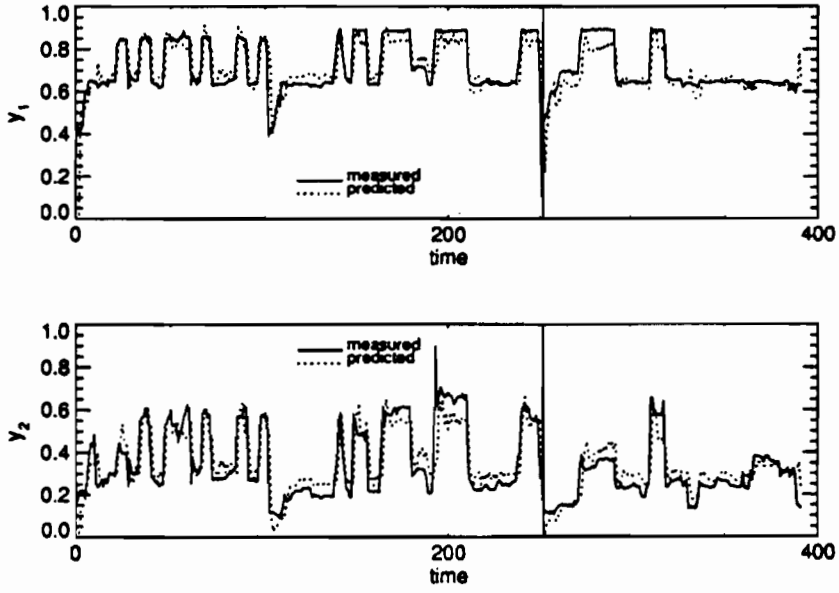


Figure 10.26 REC: long-term prediction for REF.

Table 10.4 RMS prediction errors for the wastewater plant and the catalytic reformer

(a) the wastewater plant

unit: 1/100		PRE	T12	SED
one-step ahead FFN training	NH_4^+	3.5	4.3	2.8
	NO_3^{2-}	5.2	3.8	3.9
	PO_4^{3-}	3.6	3.2	1.7
testing	NH_4^+	1.5	3.1	3.8
	NO_3^{2-}	4.1	2.4	1.7
	PO_4^{3-}	4.0	1.8	2.8
long-term FFN training	NH_4^+	45.3	12.5	31.3
	NO_3^{2-}	13.6	8.5	21.7
	PO_4^{3-}	47.0	10.0	18.7
testing	NH_4^+	43.1	10.2	23.6
	NO_3^{2-}	12.0	8.6	11.8
	PO_4^{3-}	54.8	7.4	18.5
long-term REC training	NH_4^+	4.0	4.5	8.1
	NO_3^{2-}	5.7	5.6	7.0
	PO_4^{3-}	5.1	5.3	4.2
testing	NH_4^+	4.0	9.5	11.4
	NO_3^{2-}	3.8	6.0	6.2
	PO_4^{3-}	18.8	7.0	7.0

(b) the catalytic reformer

	training		testing	
unit: 1/100	y_1	y_2	y_1	y_2
FFN (1-step ahead)	8.2	9.1	5.5	6.2
FFN (long-term)	10.4	14.5	8.6	12.5
REC (long-term)	8.6	9.9	5.6	6.3

steps (i.e., prediction horizon) was equal to the total number of samples: 900 for the wastewater system and 392 for the catalytic reformer. However, if the networks were used for model predictive control, the prediction horizon would be much smaller, e.g., twenty or thirty steps ahead into the future. It is important to know how well both networks perform within a smaller prediction horizon. In this example, the width of the prediction horizon, p, is varied from 1 to 30, and the prediction errors are calculated for p = 1, 5, 10, 15, 20, 25, and 30. The RMS prediction errors are plotted against p in Figure 10.27. Notice that the prediction error is calculated only at the end of the prediction horizon.

The RMS errors are calculated for all data points in the testing set: 300 for the wastewater system and 140 for the reformer. In order to predict the first p points of the testing set, training data that immediately precede the testing data were used for predictions. As shown in the figure, the prediction errors of a maximum likelihood network increase significantly as the prediction horizon becomes large, whereas those of a pure robust network are more stable. Although in some cases a maximum likelihood network gives results similar to a pure robust network (e.g., Figure 10.27f), the pure robust network gives consistently better results than a maximum likelihood network in cases where $p > 5$. The reason is that conventional maximum likelihood training is based on minimizing errors in one-step-ahead predictions. On the other hand, pure robust training minimizes multiple-step prediction errors.

Even though the pure robust network is normally trained over the whole trajectory of the training data, it can be trained over a smaller trajectory [47]. For example, if the prediction horizon in MPC contains p time steps, we can train the net over a trajectory of p time-steps [48]. In this case, one can set up a moving window of width p along the whole training set. The network is then trained over the trajectory within this moving window. This approach results in a case of semibatchwise backpropagation through time. A similar effect may be obtained at lower cost by using the compromise method, to be discussed in section 10.4.6.

In summary, robust training is more appropriate than conventional training for ANNs to be used in model predictive control.

10.4.5. Alternatives to the MLP in Control Theory and Neural Nets

10.4.5.1. Summary The empirical results of sections 10.3 and 10.4 were all based on the multilayer perceptron (MLP)—a very simple form of ANN. Many control theorists have argued that MLPs are essentially static models, and that true system identification requires more dynamic models analogous to those used in control theory. This argument is correct in principle; however, there are alternative forms of ANN that provide nonlinear generalizations of the usual dynamic models.

In practical applications, simple ANNs are often good enough. In some applications, simple MLPs cannot forecast $\boldsymbol{Y}(t+1)$ well enough, but the problem can be solved by a simple change of variables (e.g., forecasting $\boldsymbol{Y}(t+1) - \boldsymbol{Y}(t)$ rather than $\boldsymbol{Y}(t+1)$ [49]). This subsection will discuss alternative models and functional forms for use in applications where MLPs are *not* good enough. When an MLP model appears adequate at first, but leads to limited capabilities in control applications, then it is important to reexamine that model (as we did in section 10.3), and to consider using an alternative model or adaptation procedure.

10.4.5.2. Classical NARMAX and Kalman Models Among the models used in control theory and statistics, one of the better known is the classical NARX (*N*onlinear *A*uto*R*egressive model with e*X*ogenous variables) model, which may be written:

$$\hat{\boldsymbol{Y}}(t) = \boldsymbol{F}(\boldsymbol{Y}(t-1), ..., \boldsymbol{Y}(t-k), \boldsymbol{u}(t-1), ..., \boldsymbol{u}(t-k), W), \tag{45}$$

where k is the order of the model and W is the set of model parameters.

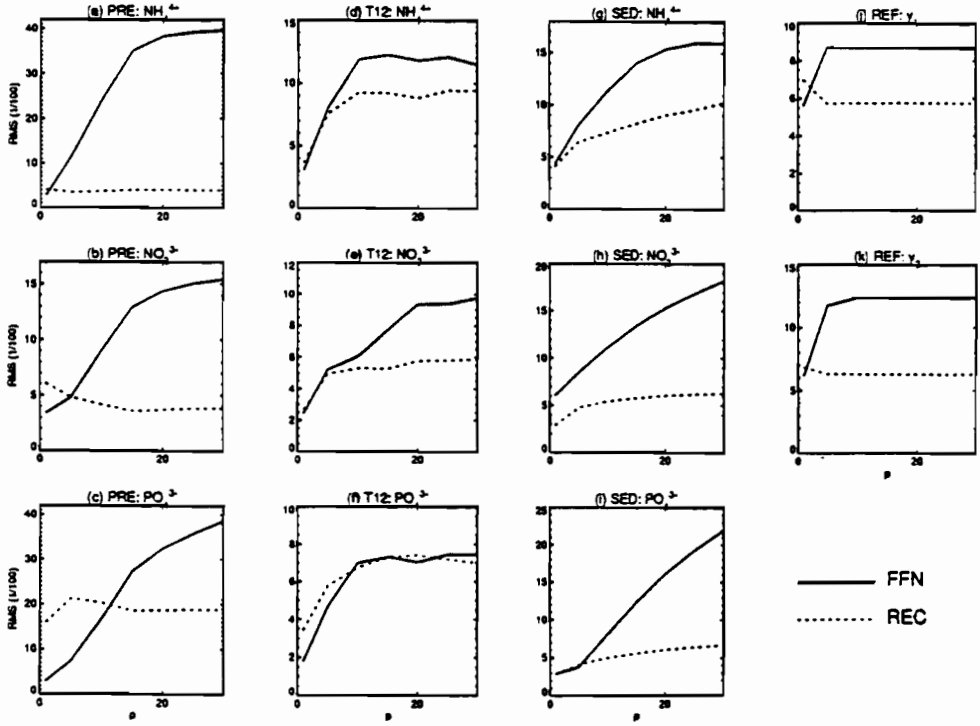


Figure 10.27 Prediction errors (RMS) vs. p for a FFN and a REC. Notice that there are 300, 300, 300, and 140 data points in the testing set of PRE, T12, SED, and REF systems, respectively. The RMS errors are calculated based on the scaled variables in all cases.

In order to represent a general NARX model, we need to use a class of models that can represent any arbitrary function $\boldsymbol{F}$; since MLPs can approximate any well-behaved vector-valued function $\boldsymbol{F}$, they can represent a general NARX model if their inputs and outputs are arranged as in equation 45. The resemblance between equation 45 and equation 20 should make it clear that MLPs can do justice to this class of model. (Strictly speaking, however, there are many nonlinear functions that can be represented more *parsimoniously* by adding a feature called "simultaneous recurrence," which is discussed in section 10.4.6.4. Simultaneous recurrence is one more technique for improving performance at the cost of doing more work, a technique that can be used to upgrade any of the models to be discussed in this section.)

In their classic introductory textbook [50], Box and Jenkins discuss a variety of *stochastic* process models, including the AR(p) model, based on a special case of equation 45:

$$Y(t) = w_1 Y(t-1) \ldots + w_p Y(t-p) + e(t), \tag{46}$$

where $e(t)$ is a random number with a Gaussian distribution. They also discuss the ARMA(p, q) model, which may be written:

$$Y(t) = w_1 Y(t-1) \ldots + w_p Y(t-p) + e(t) + \theta_1 e(t-1) + \ldots + \theta_q e(t-q). \tag{47}$$

In theory, an ARMA model of finite p and q can always be represented by an equivalent AR model *of infinite order*. Superficially, this suggests that AR models (and their nonlinear generalizations) are always good enough to model any process. However, the need for an infinite number of parameters is of enormous practical importance. In *any* form of identification or estimation, the speed of learning and the quality of generalization (i.e., the estimation errors after a finite database of experience) depend on the number of parameters to be estimated; it is important to reduce the number of parameters as much as possible. Therefore, when there is a *possibility* that the system to be identified is an ARMA process, it is important to use an ARMA specification or the equivalent.

Box and Jenkins [50] explain how an ARMA (1, 1) process results very simply from an AR (1), process observed through measurement noise, as in:

$$X(t) = wX(t-1) + e_1(t) \tag{48}$$

$$Y(t) = X(t) + e_2(t), \tag{49}$$

where e_1 represents random noise in the system to be controlled, and e_2 represents measurement error in observing that process. If the variable Y is observed but the variable X is not, then the behavior of Y over time fits an ARMA (1, 1) model (albeit with different parameters). The general state-space model of linear control theory is simply a multivariate generalization of equations 48 and 49; therefore, it leads to a vector ARMA process. This is the reason why models of the ARMA type are of great importance in control applications. In order to deal with nonlinear problems, authors like Leontaritis and Billings [51] have recently advocated the use of NARMAX models, which may be written as:

$$\hat{\boldsymbol{Y}}(t) = \boldsymbol{F}(\boldsymbol{Y}(t-1), \ldots, \boldsymbol{Y}(t-p), \boldsymbol{u}(t-1), \ldots, \boldsymbol{u}(t-p), \boldsymbol{e}(t-1), \ldots, \boldsymbol{e}(t-q)), \boldsymbol{W}), \tag{50}$$

where:

$$\boldsymbol{e}(t) = \boldsymbol{Y}(t) - \hat{\boldsymbol{Y}}(t) \tag{51}$$

is a random vector.

An alternative, equivalent way to cope with measurement noise in the linear case is to use Kalman filtering [52,53]. Instead of just modeling the observed vector, $\boldsymbol{Y}$, one may try to estimate the underlying state-space vector, which we will denote as $\boldsymbol{R}$. (In control theory, the state-space vector is usually denoted as x, but there are advantages to using a different notation with ANNs.) We arrive at a model that may be written:

$$\hat{\boldsymbol{Y}}(t) = \boldsymbol{H}\hat{\boldsymbol{R}}(t)$$

$$\hat{\boldsymbol{R}}(t) = \boldsymbol{A}\bar{\boldsymbol{R}}(t-1) + \boldsymbol{B}\boldsymbol{u}(t) \tag{52}$$

$$\bar{\boldsymbol{R}}(t) = \hat{\boldsymbol{R}}(t) + \boldsymbol{M}(\boldsymbol{Y}(t) - \hat{\boldsymbol{Y}}(t)),$$

where $\hat{\boldsymbol{Y}}$ and $\hat{\boldsymbol{R}}$ are forecasts based on prior data, and $\bar{\boldsymbol{R}}$ is our best guess as to the true value of $\boldsymbol{R}$ at time t, after the measurements from time t are available. H, A, B, and M are matrices of parameters of the model. One consistent way to estimate the parameters in this model is simply to minimize the square error in predicting $\boldsymbol{Y}(t)$. (Strictly speaking, this model does contain redundant parameters, which makes the error-minimizing values nonunique; however, this is not a problem unless there is some other reason to want to estimate specific hidden variables. For example, if one of the unobserved variables represents temperature, and high temperatures cause melting behavior outside of what the linear model predicts, one then needs to use more than just empirical data on the observed variables from normal operating regimes.) There are nonlinear extensions of Kalman filtering, but many researchers consider them cumbersome and less than completely satisfactory.

10.4.5.3. Time-lagged Recurrent Networks (TLRNs) The simple ANN models of sections 10.3 and 10.4.1—which are a special case of the NARX model—cannot effectively subsume the NARMAX or Kalman models above. However, there is a class of ANN models, developed by Werbos [21,34,54], which *can* subsume those models: the time-lagged recurrent network (TLRN). In effect, the TLRN provides an efficient nonlinear generalization of the Kalman filter, *when* our goal is to estimate the state vector or to predict $Y(t)$ over time. *Unlike* the classical ARMA models, the TLRN does not provide a general scheme for describing the *probability distribution* for *alternative* possible values of $\boldsymbol{Y}(t)$; to make that generalization, see the discussion of the Stochastic Encoder/Decoder/Predictor in Chapters 3 and 13.

In the neural network literature, a "recurrent network" refers to any ANN in which a neuron can use its own output as one of its inputs, or, more generally, in which the flow from inputs to outputs may contain loops. There are two fundamentally different *kinds* of recurrence [54], each of which has its uses: (1) *simultaneous* recurrence, discussed in Chapters 3 and 13; and (2) time-lagged recurrence. In time-lagged recurrence, we simply allow any neuron at time t to receive, as input, the output of *any* neuron at time t-1. It is trivial to allow inputs from times $t-2$ through to $t-k$ as well [22], but this does not really add any generality in practice; lags greater than 1 can be represented quite effectively by bucket brigades or exponential decays within a lag-1 structure.

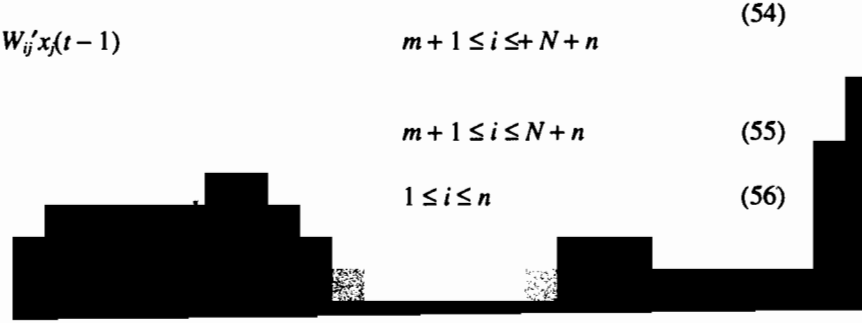
The most classical form of TLRN is a multilayer perceptron, MLP, *modified* to allow crosstime connections. The equations are:

$$x_i(t) = X_i(t) \qquad 1 \le i \le m \tag{53}$$

$$v_i(t) = \sum_{j=0}^{i-1} W_{ij}x_j(t) + \sum_{j=1}^{N+n} W'_{ij}x_j(t-1) \qquad m+1 \le i \le +N+n \tag{54}$$

$$x_i(t) = s(v_i(t)) \qquad m+1 \le i \le N+n \tag{55}$$

$$\hat{Y}_i(t) = x_{i+M}(t) \qquad 1 \le i \le n \tag{56}$$



where x_0 represents the constant 1, where $N+n$ is the total number of neurons, where $\boldsymbol{X}$ is the vector of *all* inputs at time t (usually made up of $\boldsymbol{Y}(t-1)$ and $\boldsymbol{u}(t)$), and where m is the total number of input neurons. In practice, it works best to delete (or "zero out" or "prune" most of the weights W and W', but equation 54 indicates the full range of connections that are allowed.

In this arrangement, it should be possible to represent the estimated state vector, $\hat{\boldsymbol{R}}$, through hidden nodes in the network. Any set of filtering equations could be approximated as well as desired by the subnetwork that leads up to the calculation of those nodes. One can adapt this entire system by minimizing square error, exactly as in basic backpropagation. It is straightforward to calculate the relevant derivatives (either for maximum likelihood estimation or for the pure robust method) through the use of backpropagation through time; see Section 10.8.

10.4.5.4. Sticky Neurons and Their Capabilities Time-lagged recurrence can yield two additional capabilities, which can be very important in intelligent control:

- "Short-term memory"—the ability to account for phenomena observed many time periods ago, which are not currently observable but are still currently relevant. For example, the person who turns away from a loose tiger—to run—needs to account for the presence of a tiger behind his back even though he can't see the tiger any more. Likewise, in manufacturing, short-term memory allows you to account for the properties of the last part you made even as you adapt to making a new part.
- The ability to estimate slowly varying process parameters. The need for this is one of the real motives behind classical adaptive control.

In both cases, recurrent hidden nodes can provide the required memory or parameter estimates, in principle. However, the specification in equation 55 does not provide enough stability to make this realistic. An alternative specification [35], inspired by discussions of the Purkinje cells of the cerebellum, is the "sticky neuron" defined by:

$$v_i(t) = \sum_{j=0}^{i-1} W_{ij} x_j(t) + \sum_{j=1}^{N+n} W_{ij}' v_j(t-1). \tag{57}$$

When $W_{ij}' = \delta_{ij}$, this system becomes extremely stable, stiff, or sticky. Because it is even stiffer than the usual TLRN, the user is advised—once again—to first adapt a less stiff or less general version of the network in order to arrive at good initial values for the weights. In some cases, one might initially fix the W_{ij}' to δ_{ij} or to $0.9\delta_{ij}$, and then adapt these sticky weights only after the other weights have settled down. One might use the gradient-based learning rate adjustment scheme, discussed in Chapters 3 and 13, to adapt a different learning rate for the sticky weights. For system identification applications, one might even consider a more extreme form of sticky neuron, also suggested by Werbos [35]:

$$v_i(t) = v_i(t-1) + \sum W_{ij}(Y_j(t-1) - \hat{Y}_j(t-1)). \tag{58}$$

As with other forms of TLRN, it is straightforward to adapt these networks by backpropagation.

10.4.6. Beyond the Pure Robust Method: Alternatives, Earlier Work, and Research Opportunities

10.4.6.1. Introduction The goal of this section was *not* to promote the idea of using the pure robust method in any and all problems of system identification. Up to this point, we have simply tried to prove, with real-world examples, that neural network applications can yield much more useful, practical results if we go beyond the conventional approaches to training or estimation. The pure robust method is a first step in that direction, but there are enormous opportunities available to other researchers who choose to go beyond that first step. It is very sad to see hundreds of papers in the neural network field and the control theory field that contribute relatively little either to improving our arsenal of usable, practical tools, or to deepening our fundamental understanding. At the same time, we see numerous opportunities here—largely neglected—to achieve both things at the same time, simply by paying more attention to certain basic issues.

The main goal of this subsection will be to describe these opportunities in more detail. The key to these opportunities is to *bring together* and *apply* certain ideas that already exist in various strands of control theory, statistics, neural network theory, information theory, epistemology, and elsewhere. A perfect synthesis of these ideas would lead to mathematical algorithms that are impossibly complex to apply in the real world; however, there are plenty of opportunities to proceed in a step-by-step, empirical fashion, applying one idea at a time, improving one's approximations one step at a time, as is common in neural network research and practical engineering. We ourselves do not claim to have extracted the full potential of this approach.

This subsection will proceed as follows. First, for the practical and impatient researcher, we will discuss the compromise method—one step beyond the pure robust method—and the empirical work to date on the compromise and pure robust methods. Second, we will briefly discuss the issue of adaptation in *real time*. Third, we will describe the four successive levels of theoretical analysis that were used to develop the compromise method, and their relations to other issues and methods related to intelligent control:

1. The maximum likelihood approach;
2. The Bayesian (prior probability) approach;
3. The utilitarian (loss function) approach;
4. A modified utilitarian approach.

A deeper, more rigorous reevaluation of this analysis would presumably lead to methods more general and more powerful than the compromise method. Section 10.4.6.4 will describe some preliminary efforts on these lines, and the new approaches that result from them. Likewise, there are a host of similar methods and approaches developed in the field of control theory [31] that could be reevaluated more deeply and extended to the neural network case. Statisticians often disparage the relatively ad hoc, empirically motivated methods one often finds in sources like the *Journal of Forecasting*; however, empirical results of that sort—*if* analyzed at a deeper theoretical level—could also be very useful in guiding us towards more general methods.

10.4.6.2. The Pure Robust Method, the Compromise Method, and Earlier Work The first experiments with the pure robust method were motivated by an attempt to fit a political forecasting model to empirical data back in 1972 [3]. Previous attempts using multiple regression (i.e., conventional maximum likelihood estimation and an ARX model) had led to very poor forecasting results in split-sample multiperiod forecasting. There was reason to believe that the available data on the relevant variables—nationalism and political mobilization data going back up to two centuries—might be subject to some measurement noise. Based on maximum likelihood theory, Werbos fitted a simple vector ARMA (1, 1) model—similar to equations 48 and 49, but with complex nonlinear exogenous variables added as well. Just for the sake of comparison, Werbos also deleted the process noise term (e_2) and estimated the "measurement-noise only" model—which is equivalent to the pure robust method. (The vector ARMA model was estimated by backpropagation through time.)

The surprising result was that the pure robust method cut forecast error in half, compared with the ARMA model. From a classical maximum likelihood viewpoint, this shouldn't happen. A more general model—such as the ARMA model—should do at least as well as the more specialized model, at least if the parameters were significant (they were), and if there were some reason to expect the presence of process noise (there was). The ARMA model, by contrast, reduced prediction errors by only about ten percent compared with multiple regression.

To test these results, Professor Mosteller of Harvard recommended a set of simulation studies. Werbos performed multiple simulations of two different dynamic processes, each with six different noise generators attached. The first process was:

$$X(t+1) = (1.03)X(t)(1 + e_1(t)) \tag{59}$$

$$Y(t) = X(t)(1 + e_2(t)) \tag{60}$$

The second dynamic process was similar, except that equation 59 was replaced by:

$$X(t+1) = (.38X(t) + .35X(t-1) + .3X(t-2))(1 + e_1(t)). \tag{61}$$

The six noise processes chosen for e_1 and e_2 involved Gaussian white noise, Gaussian white noise with random outliers in five percent of the cases, and dirty noise with correlations over time. Each combination of noise process and dynamic model (twelve combinations in all) was simulated ten times, yielding simulated time series of two hundred observations each. Broadly speaking, the ARMA model did indeed cut the prediction and estimation errors by a factor of two compared with multiple regression on the simulated data; however, the pure robust method cut the errors in half yet again. The dirtier the system, the bigger the differences; however, even with the simplest systems, the ranking of the methods was still the same [3].

Despite these results, it was obvious, intuitively, that the pure robust method would not be the ideal method to use under *all* circumstances. For example, in equations 59 through 61, there is an underlying exponential trend to the data, which makes it *possible* for a forecasting model to track the overall trajectory of the data over long time intervals. This is not always the case. For example, consider the process:

$$X(t+1) = X(t) + W + e_1(t)$$
$$Y(t) = (\sin X(t)) + e_2(t). \tag{62}$$

If the noise, $e_1(t)$, is small, it should be possible to track this process by using any model that is capable of outputting a sine wave—but only for a certain amount of time. As the time series grows longer, the random phase shifts will always ensure that the later part of the data will get out of phase with the earlier part, so that comparison between predicted and actual values at large times t will always show large errors.

In cases like equation 62, we do *not* need to go all the way back to the nonrobust AR (1) or ARMA (1, 1) models. For example, we can try to minimize multiperiod prediction errors, but limit ourselves to a time horizon of p—a parameter we must pick very carefully. Alternatively—and at lower cost—we can use the compromise method, proposed in [3] as a way to deal with these kinds of situations.

In the compromise method, we minimize a measure of error, E_4, which is defined by the following system of calculations:

$$\hat{\mathbf{Y}}(t) = \mathbf{F}(\hat{\mathbf{Y}}(t-1), ..., \hat{\mathbf{Y}}(t-p), \mathbf{u}, W) \tag{63}$$

$$\hat{Y}_i(t) = (1 - w_i)\hat{Y}(t) + w_i Y_i(t) \tag{64}$$

$$E_4 = \sum_t^T L(\hat{\mathbf{Y}}(t), \mathbf{Y}(t), \mathbf{w}), \tag{65}$$

where $\mathbf{F}$ represents the forecasting model or ANN, where $\mathbf{u}$ represents all the various exogenous variables at various times, where W is the set of model parameters or weights, and where $\mathbf{w}$ (made up of components w_i) is a vector of additional parameters or filtering constants. When $\mathbf{w} = \mathbf{0}$, then the compromise method reduces to the pure robust method. When $\mathbf{w} = \mathbf{1}$, it reduces to conventional maximum likelihood estimation. (If you feel nervous about your implementation of the compromise method, you can check to make sure it matches the simpler methods in those cases.) A $\mathbf{w}$ of 0.01 would, in effect, yield a delay time of seventy time periods in equation 64, and be very similar to minimizing the 70-period prediction error. If we pick $\mathbf{w}$ so as to minimize the conventional loss function:

$$L_1(\hat{\mathbf{Y}}(t), \mathbf{Y}(t)) = \sum_i (Y_i(t) - \hat{Y}_i(t))^2, \tag{66}$$

then the compromise method reduces to something like a special case of nonlinear Kalman filtering—which, by its equivalence to ARMA estimation, loses the robustness feature we are looking for. After some contorted analysis, Werbos [37] developed an alternative loss function:

$$L_2(\hat{\mathbf{Y}}(t), \mathbf{Y}(t), \mathbf{w}) = \sum_i \left(\frac{(Y_i(t) - \hat{Y}_i(t))^2}{\sigma_{Y_i}^2 (1 - |w_i|)^2} \right) \tag{67}$$

which led to superior performance on political data that were so noisy that the pure robust method did not work very well. In analyzing these results, Werbos suggested an alternative loss function:

$$L_3 = \sum_i \frac{|Y_i(t) - \hat{Y}_i(t)|}{\sigma_{Y_i}(1 - |w_i|)}. \tag{68}$$

In these tests, w_i was restricted to the interval (0, 1).

The model used in these tests was an upgraded version of a model then used by the Joint Chiefs of Staff for global long-range strategic planning [37]. The compromise method with the alternative loss function led to a reduction of multiperiod prediction error that was only about twenty-five percent for the political conflict variables (such as revolutions in Latin America), which were extremely difficult to predict in any case. It led to a fifty percent reduction in error in predicting GNP, both in the Latin American study and in a follow-up study of U.S. economic data (based on a simple model of the U.S. economy developed by Kuh). Several unpublished student papers at the University of Maryland showed similar reductions of error.

The biggest advantage of the compromise method is its ability to shift smoothly—to a *different* extent for *different* prediction targets—between the extremes of conventional adaptation and pure robust adaptation. In conventional statistics, it is often good enough to give the user two extreme estimation methods and provide guidelines for which method to use when [56]. In intelligent control—where many variables need to be predicted, and the decision needs to be automated and changeable over time—something like the compromise method is essential. *This kind of smooth variation between a classical approach and a worst-case or robust approach can be implemented on many levels in the design of a robust system.*

The loss functions in equations 67 and 68 appear useful, according to the empirical results so far. However, no one has attempted a neural network application as yet. Because of the gross approximations made in deriving these loss functions, there is every reason to believe that you, the reader, could come up with something similar but better—especially if you revisit the theoretical issues to be discussed below. On the whole, equations 67 and 68 appear very effective in nudging the w_i parameters closer to 1 when they are already close to 1; however, they do not have such a strong effect on values of w_i which are close to 0. Also, it may be useful to weight the errors for different variables in a different way; for example, for models embedded within an adaptive critic control system, one might weight the error in predicting Y_i according to the RMS average of the derivative of J with respect to Y_i.

10.4.6.3. Real-time Learning Section 10.8 will describe how to implement the compromise method in an off-line mode, using backpropagation through time. *All* of the models and identification methods in this chapter boil down to minimizing an error function over time, in the presence of time lags. To implement these methods, one basically needs to calculate the derivatives of error with respect to the weights or parameters. Therefore, for real-time adaptation, one basically needs a method that moves *forwards* in time, to calculate the same derivatives that are normally calculated by backpropagation. (In some real-time applications, it may be useful to develop an off-line model first, in order to develop useful hidden variables; these variables, in turn, can be input to a more static real-time network.)

One can calculate the derivatives exactly in forwards time by using the forwards perturbation technique [20], which was later called the "Williams-Zipser" method and "dynamic backpropagation"; however, the resulting calculations are N times as expensive as backpropagation through time. If N is small, this may be useful. If it is large—as in biological systems—the method becomes hopelessly unrealistic. See Chapter 5 for more details. Section 10.6 may help in using [20].

Alternatively, one can simply truncate the derivatives, as suggested by equation 25; however, this dramatically reduces robustness over time because it totally ignores the connections over multiple time intervals.

Finally, one can approximate the derivatives in a consistent way by using the Error Critic design, presented in Chapter 3. The present chapter gives a variety of error functions different from the example in that chapter, but the extension to alternative error functions and functional forms is straightforward. We would speculate that the human brain itself is based on such an extension.

10.4.6.4. Basic Principles and Derivation of the Loss Function for the Compromise Method

Once again, the compromise method evolved in response to empirical results. The approximations used to derive it were very drastic and can be improved upon (or made more adaptive). Before understanding the robust approach, theoretically, it is necessary to review some earlier approaches, which will still be important to future research.

The early work by Werbos [3] was initially based on the classical Bayesian approach to parameter estimation. In that approach, one begins by formulating a *stochastic* model, such as equations 48 and 49 or equations 50 and 51. (Nearest-neighbor systems or spline models can also be formulated as stochastic models, but it takes some care to avoid logical inconsistencies.) One tries to find that set of parameters, W, that have the highest probability of being the true set of parameters, based on what we know *after* accounting for the available data; in other words, we try to maximize the conditional probability $Pr(W \mid data)$.

The first deep problem for theorists in parameter estimation is the fact that $Pr(W \mid data)$ is not really well defined in an objective manner. From Bayes' law, it is well known that:

$$Pr(W \mid data) = \frac{Pr(data \mid W) Pr(W)}{Pr(data)}. \tag{69}$$

On the right-hand side, only one term, the "likelihood term"—$Pr(data \mid W)$—is well specified, for any well-specified stochastic model. We can ignore the term $Pr(data)$ because it does not affect our maximization with respect to W. The third term—$Pr(W)$—represents our *prior* assessment of the probability of different parameters, our assessment *before* any data become available. No matter how large our database, this term is still present, and it has been a source of deep and persistent confusion even in practical applications.

In the maximum likelihood approach—as practiced by most engineers—one tries to be "scientific" by assuming that all sets of parameters are equally likely *a priori*. This simply wipes out the $Pr(W)$ term. By maximizing the likelihood term itself, one has a direct recipe for calculating what appears to be an optimal estimate of all parameters. Given a perfect model and an infinite database (with enough variety of the input variables), this approach will eventually converge on the optimal estimates.

Practical users of statistical methods, beset by limited data, usually recognize the need to account for specific, prior information they have about the variables they are trying to predict. For example, econometricians have long grappled with the problem of how to reconcile empirical, statistically based models with first-principle, engineering-based models [56]. Typically, econometricians do *not* try to quantify their prior probabilities, $Pr(W)$, and use a Bayesian regression routine. They use their prior information to guide them in setting up alternative stochastic models; they use it again, *after* performing maximum likelihood estimation, to select the final model and to "tweak" its parameters [56]. Methods like backpropagation could be used as part of this process to adapt or calibrate parameters in complex first-principle models in a more automated way.

There is *another* kind of prior information, above and beyond our knowledge of specific variables, that is extremely important to fields like neural networks and intelligent control. To explain this crucial point, we will begin with an example.

In the neural network field, we often try to "learn" the mapping from a vector X to a scalar Y. Suppose, for example, that the vector X had twenty different components, and that we can distinguish ten possible values for each of these twenty variables. Thus, to specify Y as a function of X, we would need to know the values of Y for each of 10^{20} different points. Even if we do *not* assume the possibility of noise, we would need a huge amount of data even to begin learning the mapping. It is this kind of impossible-looking problem that makes the purist variety of theoretician give up and hide, thereby leaving the deep and important theoretical questions to people like engineers.

To cope with these kinds of practical problems, we need to exploit the fact that *not all functions are equally likely a priori*. The real world tends to have certain properties of smoothness, of similar values in nearby points, of sparse connectivity, of symmetry, of global order, and so on. In the real world, simpler models (or, equivalently, models with a lot of the parameters zeroed out) are more likely to be true or robust than are complex models. Deep philosophers have recognized for centuries that this principle—Occam's Razor—is essential to human learning and induction. (Indeed, one can argue that Occam's work had a very substantial effect on subsequent human history, particularly on science and religion.)

Our first deep challenge, then, is to formulate Occam's Razor in a more operational way, which is truly useful in adapting ANNs (or other models) in the real world. The Razor must be sharp enough to prevent floundering with overly complex models, but dull enough to allow our system to be truly "open-minded"—able to adapt to a wide variety of possible environments. Good ANN researchers already know that simpler nets lead to better generalization, but we need to make this knowledge firmer and more operational.

This challenge was met successfully in the field of classical AI by the work of Solomonoff in the early 1960s [57]. Solomonoff proposed that the *a priori* probability for any model should be e^{-kS}, where k is some constant and S is the number of symbols required to describe the model *as a program in some kind of basic Turing machine*. Because any Turing machine can be programmed to mimic any other kind of Turing machine, with a *finite* program (using, say, S_0 symbols), we are *guaranteed* that such a system could "learn" to change its language after acquiring only S_0 pieces of information. In effect, Solomonoff proved that such a system would be invariant to the initial choice of modeling language; thus, it would be truly open-minded in an important way. Solomonoff's work is of enduring importance, even to ANN research, but—like much of classical AI—it mainly focuses on *Boolean variables*. For our purposes, we need to extend the concept further to deal with models of continuous

variables. We need to account more concretely for phenomena like local smoothness, forecasting by analogy, and so on.

In neural network theory, it is important that the multilayer perceptron (MLP) does *not* live up to Solomonoff's notion of a Turing machine or Turing language. A Turing machine has the ability to write out to a kind of "blackboard" or "tape," which it can later read back and reanalyze, all as part of the calculations it makes before outputting a single forecast. The simultaneous-recurrent network does have the ability to perform multiple calculations, inputting its own outputs, as part of a single computation cycle (see Chapter 13); therefore, it is very close to fulfilling Solomonoff's requirement. It is not quite perfect, because the number of cells in the network is finite, whereas a true Turing machine is allowed an infinite blackboard. To *completely* live up to Solomonoff's results, a system must be able to engage in overt symbolic reasoning with the aid of unlimited paper and computer support; this kind of symbolic reasoning can be *learned* by neural networks (e.g., the networks in the brain), but it may well be impossible to implement it on a lower level as part of the neuronal hardware [15].

When we insert the Solomonoff priors back into equation 69, and take logarithms on both sides, we end up with an interesting recipe for parameter estimation. We are asked to minimize the sum of two terms: (1) the log likelihood, which can be described as the residual entropy of the database after the model is available; and (2) the kS term, the entropy or information content of the model. This provides a rational basis for the notion of minimizing "total entropy" or "total complexity," a well-known approach in information theory. Theorists like Solla [58] and Kazakos [59] have tried to apply this approach in the neural network field; however, Kazakos has stated that there are many insights and ideas about complexity measures and robustness in journals like the *IEEE Transactions on Information Theory* that have yet to be fully used or appreciated in system identification (e.g., see [58]).

Economists and statisticians have also developed formulations of Occam's Razor for use in the linear case. Economists often find themselves with data on a very limited number of time points, for a very large number of variables. They have been forced to develop ways of coping with this problem. There are two major techniques in common use:

1. Ridge regression, which effectively penalizes large estimates of parameters
2. Stepwise regression (by hand or automated), in which parameters are deleted (zeroed out) when they fail a basic test of "statistical significance"

The well-known statistician Dempster [60] has given a thorough review of these techniques and their many (less effective) competitors. Most of them have analogues in the neural network literature.

In ridge regression, one assumes that the *parameters themselves*—W—come from some sort of Gaussian distribution. Instead of assuming *a priori* that the variance of that distribution is *infinite* (as in maximum likelihood theory), one tries to *estimate* the variance of this distribution. This is called the "empirical Bayes" approach. Clearly, this general flavor of approach could be used more generally to help us generate prior probabilities of ANN adaptation. In practice, ridge regression asks us to minimize an error function:

$$\sum_t (e(t))^2 + k \sum_{i,j} (W_{ij})^2 , \tag{70}$$

where k is another parameter to be estimated in another way [60]. It is straightforward to apply this method directly in ANN research (as first suggested in [34]), but it may be possible to do better yet. One can argue that equation 70 is too "closed-minded," in a sense, about large parameters; that it does not adequately value the zeroing out of parameters (which is also important [60]); and that it does not reflect the tendency for the weights affecting a given target to vary considerably in size. To account for such concerns, one might, instead, assume a *lognormal* probability distribution, which leads to an error function:

$$\sum_t (e(t))^2 + k \sum_{i,j} \log \frac{1}{|W_{ij}|} . \tag{71}$$

This kind of error function reflects the usual practice in econometrics of zeroing out parameters that are only one standard deviation or so away from zero. The constant k can be adapted by techniques similar to those of Su, described earlier in this section. Because the term on the right has a powerful effect in zeroing out some weights, one must take care in developing initial weights, as with the other methods in this chapter. In the limit [34], one might give greater credence (a smaller k) to connections between neurons in similar "clusters," etc. In the ANN field, there is recent work on input compression, by Cooper, and on B-spline methods, which may be significant and novel here.

In any event, the statistical profession has firmly established the validity of minimizing modified error functions that include a direct reference to the parameters or weights of a model.

Unfortunately, the sophisticated Bayesian or Occamite approach above was still not enough to explain the empirical results discussed in section 10.4.6.2. ARMA models were estimated with significant t-ratios and F-ratios, which suggested that the full model should be used in forecasting; however, the pure robust method still did better, both in multiperiod forecasting *and* in parameter estimation, with empirical *and* simulated data.

To explain or exploit these empirical results, one must move up to a more sophisticated theoretical approach—the "robust estimation" approach [30].

In robust estimation, we give up the assumption that our original stochastic model will in fact be "true," in some objective sense, for some set of values for the parameters W. We assume that our stochastic model will typically (though not always) be a highly simplified description of a more complex reality, regardless of whether that model is an ANN or a first-principle model. We try to adapt or estimate that model so that it will do a good job of forecasting (or control) *despite* that limitation.

Some theoreticians have argued that robust estimation should never be necessary. If our stochastic model is not good enough—as determined by *any* objective measure of how the model fits the data—we should simply change it until it *is* good enough. In actuality, the real process we are trying to identify may simply be too complex to make this a workable strategy. If there are *many, many* objective measures available, the complexity required to account for all of them may make it difficult to avoid large estimation errors. Still, this approach should not be discarded altogether. In fact, the pure robust method could be viewed as one way of *implementing* that approach. The pure robust method may be seen as a kind of complex objective measure or diagnostic used to evaluate a stochastic model; if we find that it is *possible* to do significantly better in multiperiod forecasting than the original model suggested, we may then choose to stay with the pure robust (or compromise) method. This kind of reasoning may well be useful as a guide to tuning the filtering parameters, w_i , in the

compromise method or in various extensions of the compromise method. Still, we have to ask *what to do* when our diagnostic does tell us that better long-term forecasting is possible. Direct use of the robust methods is the obvious choice. We wouldn't want to add a lot of terms to the old stochastic models to make them fit better, simply because those models did, in fact, forecast poorly over long time intervals. There may be other possible responses, however, that may merit further research.

In robust estimation, our biggest challenge is to find workable ways of estimating a model that do not assume that its stochastic version is true. There is a huge literature on how to do this when a stochastic model is *almost* true, when we assume that the equations of the model are true *but that* the noise may not come from a Gaussian distribution. Those methods still assume that the noise is "clean" (i.e., uncorrelated). For the more general case, we are aware of only two workable approaches to robust estimation: (1) a utilitarian-based approach; (2) an approach based on minimizing parameter errors. From an Occamite point of view, the best procedure would be to always consider the set of *all possible* models, instead of just one model; however, even if we do maintain a handful of alternative models, we still need to use robust estimation techniques for each of that handful. (In effect, each explicit model must be used to integrate or represent a large set of models we cannot afford to be explicit about.)

The utilitarian approach has been around for a very long time. As an example of this approach, suppose that we are trying to forecast a single variable. Suppose that we are trying to predict the savings rate, s, of different households, as a function of income and other variables. The savings rate is defined as total savings (S) divided by disposable income. We expect that the noise in this process will not be a function of income or other variables; thus, for people earning \$1,000,000 per year, our forecast may be off by ten percent or so—\$100,000—but for people earning \$10,000 or so we expect errors on the order of \$1,000. A maximum likelihood statistician would tell us to minimize the error function:

$$E = \sum_t (s(t) - \hat{s}(t))^2. \tag{72}$$

A utilitarian economist might instead try to minimize the loss function:

$$L = \sum_t (S(t) - \hat{S}(t))^2. \tag{73}$$

The utilitarian would argue that the *cost* of an error (in many applications of this model) would depend on the *size* of the error in dollars; this justifies giving greater weight to observation where more dollars are involved. He would argue that there are two sets of parameters out there in the real world—the parameters that minimize E and the parameters that minimize L; for his application, it is L that matters, so he prefers to use a procedure that is *consistent* with his application. A maximum likelihood purist would reply that there is only *one* true set of parameters; minimizing equation 73 is simply a less *efficient* way of estimating the same set of parameters, which can be estimated more *efficiently* by minimizing equation 72. In an ideal world, one might try it both ways, and then try to see if there really were a measurable difference between the two sets of parameters; however, this judgment would be affected somewhat by one's *prior* expectations about the possibility of such a difference.

There is a simpler example of this same principle, which has long been familiar to statisticians. Suppose that we are trying to estimate the median of a random distribution. To be perfectly *consistent*, we can simply use the median of our sample data. However, if we have good reason to believe that our data comes from a Gaussian distribution, we can use the sample *mean*; this is a more efficient estimator, and we know that the median is the same as the mean anyway, *if* we are sure of the distribution. Using the sample median is essentially the robust or utilitarian way to go. There are other possible estimators, of course.

In our application, the utilitarian approach seems very straightforward, at first. In Model-Predictive Control, our payoffs depend on the multiperiod forecasting ability of our model or ANN. The pure robust method minimizes those errors directly. Therefore, it is consistent with what we need, and we should use *it* rather than the maximum likelihood method.

In actuality, there are two important complications here. First of all, our errors in control do not depend in such a simple way on the sum of prediction errors over all possible prediction intervals; the literature on control theory [31–33] suggests a more complicated relation, related to our earlier suggestions for a moving-window approach and for the compromise method. Secondly, our earlier example of a phase-shift process shows that the pure robust method may in fact be *extremely inefficient* in estimating parameters in some situations. Loss functions were developed for the compromise method in 1977 [36] that failed very badly in empirical tests, derived from a pure utilitarian approach.

The alternative loss functions shown in section 10.4.6.2 were derived from a modified version of the utilitarian approach [61]. The idea was to minimize the long-term prediction errors, *as predicted* by the model itself, but *assuming the worst* about the accumulation (i.e., autocorrelation) of error. There were gross approximations involved, but the method has proven useful.

To begin with, Werbos [61] considered the case of a simple univariate forecasting model with the compromise method, deriving:

$$Y(t) = \hat{Y}(t) + e(t) \tag{74}$$

$$\bar{Y}(t) = (1-w)\hat{Y}(t) + wY(t) = \hat{Y}(t) + we(t) \tag{75}$$

$$\hat{Y}(t+1) = \theta\bar{Y}(t) = \theta\hat{Y}(t) + \theta we(t) \tag{76}$$

$$Y(t+1) = \theta\hat{Y}(t) + \theta we(t) + e(t+1). \tag{77}$$

And likewise:

$$Y(t+n) = \theta^n\hat{Y}(t) + e(t+n) + \theta we(t+n-1) + \theta^2 w^2 e(t+n-2) + \ldots. \tag{78}$$

The error terms on the right-hand side of equation 78 represent the errors in multiperiod prediction. Assuming the worst about the correlation between errors over time, but assuming that θw is less than one (as it would be in any stationary process, when we enforce the rule that w can be no larger than one), the expected value of the square of the sum of these error terms will be:

$$\frac{\sigma_e^2}{(1 - |\theta w|)^2} \tag{79}$$

where σ_e^2 is the variance of the error term, $e(t)$. For a simple univariate model, we would then pick w so as to minimize this measure of error. For a multivariate model or an ANN, where there is no scalar θ available, we assumed that errors also accumulate across variables in the worst possible way, and assumed a θ of 1; this led to the loss functions of section 10.4.6.2. Again, one could easily do better.

Starting in 1981, we have explored yet another fundamental approach, mentioned briefly in [35]. This approach may be called the σ_W approach. In the σ_W approach, we go back to the maximum likelihood idea that there is only *one* set of parameters that is best, both for short-term forecasting and for long-term forecasting. Clearly, this will need to be reconciled with the utilitarian approach, eventually, but so far it seems to follow the empirical evidence a bit better; in predicting conflict in Latin America [37], Werbos found that the compromise method did better in *all* prediction intervals in split-sample tests. In a sense, this is like the usual robust approach, where we accept the model at face value, and assume that the problem lies in the dirty noise; however, in this case, we allow for correlated and truly dirty noise, not just non-Gaussian distributions.

The idea is to pick those values of the filtering parameters, w, which lead to the smallest possible estimation errors in those parameters—W—which will actually be used in the control application. Instead of using the maximum likelihood estimates of the estimation errors, we would like to use more robust estimates, such as the bootstrap estimates of Friedman or the empirical estimates of Hjalmarsson and Ljung [62]; however, we need to use simple approximations, in the end, to fit with the neural network rule of limiting ourselves to inexpensive calculations.

Once again, we have begun our analysis in the univariate linear case. In that case, the usual measure for standard error may be written as:

$$\sigma_\theta^2 = \sigma_e^2 \Big/ \left(\frac{\partial^2}{\partial \theta^2} \sum_t (e(t))^2 \right) \tag{80}$$

This is similar to equation 79, except that the denominator is somewhat different; thus, our prior method was somewhat similar to picking w to minimize σ_θ^2. To evaluate the denominator, note that:

$$\frac{\partial^2}{\partial \theta^2} \sum_t (e(t))^2 = \sum_t e(t) \frac{\partial^2}{\partial \theta^2} e(t) + \sum_i \left(\frac{\partial}{\partial \theta} e(t) \right)^2. \tag{81}$$

Note that the leftmost term on the right-hand side represents the correlation between error and another quantity; in general, there is little reason to expect a strong correlation, and—in a forecasting situation—we always try to estimate parameters so that previous variables correlate very little with error at time t. Thus, for a crude, preliminary analysis, we will approximate that term as zero; we will use the rightmost term of equation 81 as our denominator.

Simple substitutions yield:

$$e(t) = Y(t) - \hat{Y}(t) \tag{82}$$

$$\hat{Y}(t) = \theta\bar{Y}(t-1) = \theta(1-w)\hat{Y}(t-1) + \theta w Y(t-1). \tag{83}$$

Mathematical induction on equation 83 yields:

$$\hat{Y}(t) = \theta w(Y(t-1) + \theta(1-w)Y(t-2) + \theta^2(1-w)^2 Y(t-3) + ...). \tag{84}$$

If we assume the worst about correlations between Y values over time, we arrive at the following estimate of the RMS average size of equation 84:

$$\sigma_{\hat{Y}} \approx \frac{\theta w}{1-\theta(1-w)}\sigma_Y. \tag{85}$$

Going back to equations 81 and 82, we may approximate our denominator as:

$$\sum_t \left(\frac{\partial}{\partial\theta}\hat{Y}(t)\right)^2 \approx \sum_t \left(\frac{\partial}{\partial\theta}\left(\frac{\theta w}{1-\theta(1-w)}\sigma_y\right)\right)^2, \tag{86}$$

which reduces to:

$$\left(\frac{w\sigma_Y^2}{(1-\theta(1-w))^2}\right)^2. \tag{87}$$

Substituting this into equation 80, and ignoring the σ_Y^2 term (which has no effect on our choices of w and θ), we arrive at a different kind of loss function to minimize:

$$L_5 = \left(\sigma_e \frac{(1-\theta(1-w))^2}{w}\right)^2. \tag{88}$$

In cases where θ is near zero, and larger values of w lead to smaller σ_e, this tells us to pick w was large as possible; since $w = 1$ is the largest allowed value (representing the maximum likelihood case), we pick that value. In fact, this still applies for *any* value of θ less than one-half. However, when θ is near 1, this tells us to minimize $\sigma_e w$, which tends to push very strongly for the pure robust method.

This analysis gives us considerable insight into the proper use of the compromise method. However, it is not clear what to do about this insight in neural network applications. As a first crude approach, we might "approximate" θ_i as the square root of $1-(\sigma_e^2/\sigma_Y^2)$, as suggested in [61]; however, this is so crude that its value is questionable at present. A very crude, multivariate linear version of the reasoning above leads to a denominator more like:

$$Tr((I-\theta(I-W))^{T^{-2}}\, W^T Q W\, (I-\theta(I-W))^{-2}\, Cov(Y)) \tag{89}$$

where θ is now a matrix, where W is a filtering matrix, where Q is a positive metric representing the cost of different kinds of error, and $Cov(Y)$ is the variance-covariance matrix of Y, assuming all variables have been reduced to a mean of zero. If θ is close to being *any* kind of unitary matrix, then the optimal value of $I - W$ is not I (as in the pure robust method), but θ^{-1}. This suggests two approaches to getting maximum robustness from a complex model (such as an ANN model): (1) actually use a complete filtering *matrix* or network; (2) try to transform the representation, so that individual variables tend to represent eigenvectors of the dynamic matrix θ or other such stable variables.

Both of these approaches have some hope for them, but are completely new and unexplored in this context. The network-filtering approach leads to the obvious problem of how to minimize a rather complex error function. With simultaneous-recurrent circuits looping around a network, it is feasible to calculate inverses (or, more precisely, to multiply the inverse of a network by a vector), but there is some major complexity here and issues about numerical stability and consistency. The eigenvector approach would be very similar in spirit to the encoder/decoder schemes that are widely used in neural network research; for example, see the Stochastic Encoder/Decoder/Predictor design in Chapter 13. Likewise, variables that have big J derivatives (in an adaptive critic design) also tend to be relatively stable variables. In a network with sticky weights (as discussed in section 10.4.6.2), the sticky weights themselves may be used as estimates of θ. Clearly, it will take some creativity to get all these pieces together in a logical manner, but the pieces do exist.

Taking this all a step further, the discussion of "chunking" in Chapter 13 does include a discussion of further issues of relevance to robustness. Likewise, it would seem logical to modify our estimation of some components of W—such as sticky weights— in order to improve *their* influence on estimation errors of *other* components of W.

There is a rich field here for future research—both empirical and theoretical—of enormous importance to practical success in intelligent control.

10.5. A METHOD FOR PRUNING ANNS

Stringent pruning is often vital to ANN performance, for reasons discussed at length in this chapter. At least two other pruning or sizing methods were discussed in section 10.4. The method described here was developed in 1990 by Bhat and McAvoy [2]. This method has been applied successfully to real-world problems involving felt design; however, [2] mainly discusses applications to simulated systems and to the pH/CSTR example of section 10.3.

In reviewing the background, Bhat and McAvoy stressed the need for obtaining a good training set as well, using classical notions of experimental design [63] to ensure variance of the inputs. They reviewed earlier methods, based on deleting *entire neurons*, which have a number of limitations and, in any case, do not provide all the power we need by themselves.

The Bhat/McAvoy algorithm uses *two* evaluation functions for the ANN, one of them (L_W) used to adapt the weights, and the other (L_{NN}) used to evaluate the ANN after training. They define:

$$L_W = \left(\sum_i^n (\hat{Y}_i - Y_i)^2 \right) + \lambda * COMPLEXITY \tag{90}$$

where:

$$COMPLEXITY = \sum_k N_k \sum_j^{N_k} W_{jk}^2 \tag{91}$$

and N_k is the number of active, nonzero weights arriving at each node; and (for the case $n = 1$):

$$L_{NN} = \sigma_e^2 + \sigma_Y^2 \frac{k}{T} \tag{92}$$

where T is the number of observations in the training set, k is the number of weights in the network, and σ_e^2 and σ_Y^2 are the sample variances of the error and target variables, respectively. (The sample variance of Y is defined as the average observed value of $(Y - \mu_Y)^2$, where μ_Y is the mean of Y.)

The algorithm itself is similar to an earlier one proposed by Niida et al. [64]:

1. Start with a complex MLP full of weights and adapt the weights so as to minimize L_W with $\lambda = 0$.
2. Increment λ by a small amount, and adapt the weights again to minimize L_W.
3. Delete (strip) all the weights in the dead zone around zero.
4. After removing the stripped weights (i.e., holding them to zero), adapt the network again with $\lambda = 0$.
5. Evaluate L_{NN} for the resulting network (and store L_{NN}).
6. If the current L_{NN} is smaller than that obtained before, with a smaller value of λ, go back to step two and try again.
7. If the current L_{NN} is larger, then go back to the previous version of the ANN (for a smaller λ), and use that as the final version. Quit.

Niida et al. had proposed a different measure of complexity:

$$COMPLEXITY = \sum_k \left| \prod_j W_{jk} \right|,$$

but this failed in all of our tests because of high sensitivity to changes in λ, difficulties in handling weights that vary by orders of magnitude, and erratic behavior when small weights are deleted. The measure of L_{NN} used here was based on work by Barron for models that are *linear* in their coefficients [65]; therefore, there may be some room for improvement in the procedure.

10.6. HOW TO USE THE CHAIN RULE FOR ORDERED DERIVATIVES

The chain rule for ordered derivatives ("backpropagation") is a general-purpose tool for calculating derivatives efficiently through *any* nonlinear, sparse system. As a rule of thumb, if it takes N calculations to run a model, it should take only kN calculations to extract the derivatives of any target variables (such as error or utility) with respect to *all* of the inputs and parameters of the model, in a *single* sweep backwards through the model. As computers grow more powerful and the problem addressed grow correspondingly larger, the efficiency of derivative calculations will become even *more* important, because the ratio between N calculations and N^2 calculations will grow larger.

This section will *only* discuss the problem of calculating derivatives through an *ordered* system. Roughly speaking, an ordered system is one where we have a definite, fixed procedure for calculating the outputs as a function of the inputs; it differs from a "relaxation system," which requires the solution of a set of nonlinear simultaneous equations in order to calculate the outputs. Chapter 13 describes how to calculate derivatives effectively through a relaxation system (a "simultaneous-recurrent network"), *once* you know how to 49-calculate through an ordered system.

The section will proceed as follows. First, it will describe what an ordered derivative is and give the chain rule. Second, it will give an example of how to apply it. (The procedure here could easily be automated.) Finally, it will describe the notion of a "dual subroutine," which is crucial to the use of this chain rule within larger, more sophisticated control applications. The material here is mainly taken from [4] and [22].

10.6.1. Ordered Systems and Ordered Derivatives

An ordered system is a system made up of variables or quantities that we can calculate in a definite order. The two most common examples are static systems and dynamic systems. An example of a static system might be:

$$z_2 = 4 * z_1 \tag{93}$$

$$z_3 = 3 * z_1 + 5 * z_2. \tag{94}$$

More generally, a static system might be:

$$\begin{aligned} z_{m+1} &= f_{m+1}(z_1, \ldots, z_m) \\ z_{m+2} &= f_{m+2}(z_1, \ldots, z_{m+1}) \\ &\vdots \\ TARGET = z_N &= f_N(z_1, \ldots, z_{N-1}), \end{aligned} \tag{95}$$

where z_1 through z_m are the parameters and inputs, and where f_{m+1} through f_N are differentiable functions. When we are seeking the derivatives of a particular quantity, z_N, we usually describe the

system so that z_N is the *last* quantity to be calculated; it is convenient to give that quantity a special name ("*TARGET*").

In some ordered systems, we can still calculate the desired output (z_N) after changing the order of calculation; however, this merely gives us a choice of alternative ordering schemes, each of which is a valid starting point. (For those who are interested in parallel computing, however, it may be interesting to note that our derivative calculations will preserve the lattice structure—the parallelization—of the original model.) The *key requirement* is that we can calculate any quantity, z_i, *when* we know the values of preceding quantities.

An example of an ordered dynamical system would be:

$$Y_2(t) = W_1 * Y_1(t) + W_2 * Y_1(t-2) \tag{96}$$

$$Y_3(t) = 3 * Y_1(t-1) + 4 * Y_2(t) * Y_2(t-1) \tag{97}$$

$$E(t) = E(t-1) + (Y_3(t) - Y_3^*(t))^2, \tag{98}$$

where $E(t)$ is a kind of error measure. In actuality, this dynamic system may be treated as a *special case* of equation 95, where we define z_1 as W_1, z_2 as W_2, z_3 as $Y_1(1)$, ..., z_{T+2} as $Y_1(T)$, z_{T+3} as $Y_2(1)$, z_{T+4} as $Y_3(1)$, and so on. Using this kind of approach, Werbos has applied backpropagation even to models with *two* time dimensions, as required by some economic models of the natural gas industry [4]; however, this section will not give explicit attention to that kind of model.

The chain rule for ordered derivatives may be written (for the system in 95):

$$\frac{\partial^+ TARGET}{\partial z_i} = \frac{\partial TARGET}{\partial z_i} + \sum_{j=i+1}^{N-1} \frac{\partial^+ TARGET}{\partial z_j} * \frac{\partial z_j}{\partial z_i}, \tag{99}$$

where the derivatives with the + superscript represent *ordered* derivatives, and the derivatives without superscripts represent simple partial derivatives. To use this chain rule, you need to know only two facts:

1. That the ordered derivatives of *TARGET* with respect to an input or parameter do, in fact, represent the effect of changing that input on changing *TARGET*; these are the appropriate derivatives to use in error minimization, utility maximization, and so on.
2. That the "simple partial derivatives" in equation 99 are worked out simply by differentiating *that equation in the model* that defines z_j (or *TARGET*) as a function of earlier variables.

The full definition and proof of equation 99 was given in [3] and reprinted in [4]. To use equation 99 in practice, you must start with the case $i = N - 1$ and work backwards to $i = 1$.

It is usually best to *test* these derivatives by perturbing inputs and making sure that the calculated derivatives are right. When they aren't, one can narrow down the problem by testing the validity of *intermediate* ordered derivatives. One can perturb an intermediate quantity (or equation) z_j, and *then run* the rest of the model (from $j + 1$ to N) in normal fashion; this should change z_N in proportion to

the ordered derivative of z_N with respect to z_j. If it does not, you then know that there is a downstream problem, in calculating the ordered derivatives of z_N with respect to $z_j, z_{j+1} \ldots z_{N-1}$.

In the example of equations 93 and 94, the simple partial derivative of z_3 with respect to z_1 (the *direct* effect of changing z_1) is 3. The ordered derivative (the *total* effect) is 23, because of the *indirect* effect by way of z_2.

10.6.2. Implementation Procedure

Section 10.6.1 includes all the information you need, in principle, to apply the chain rule for ordered derivatives. However, there is a straightforward way of applying that chain rule, given at length in section IV of [4], which could be used by people who find it difficult to understand section 10.6.1. This straightforward procedure is useful even for the expert in disentangling complex systems.

In this procedure, we begin by defining an abbreviation:

$$F\_z_i \triangleq \frac{\partial^+ TARGET}{\partial z_i}. \tag{100}$$

In most applications, this abbreviation gives us all the information we need, because there is only one *TARGET* variable (like error). Also, the abbreviation can be used directly to naming variables in computer code. In more complex applications, this kind of abbreviation is even more essential. Intuitively, F_z_j may be thought of as "the feedback to variables z_j."

In this notation, equation 99 may be written:

$$F\_z_i = \frac{\partial^+ TARGET}{\partial z_i} + \sum_{j=i+1}^{N-1} F\_z_j * \frac{\partial z_j}{\partial z_i}. \tag{101}$$

The summation on the right-hand side appears complex at first; however, since the equation determining z_j will usually refer to only a few other variables z_i, the simple partial derivative of z_j with respect to z_i is zero in most cases. In most cases, the summation boils down to a sum of only one or two nonzero terms.

Step one in our procedure is to *spell out, in proper order, all of the calculations* used to calculate *TARGET*. Equations 96 through 98 are an example, if we add just one more equation:

$$TARGET = E(T). \tag{102}$$

Because dynamic systems are harder to work with than static systems, we will focus on that example of an ordered system in this section.

After spelling out the forwards system (equations 96, 97, 98, and 102), our job is simply to work out the backwards equations that will be used to calculate the required derivatives. In working out these equations, we first write down the equation for $F\_(Z)$, where Z is the *last* variable calculated in the forwards system; then we work out the equation for the variable before that, and so on. By writing down the equations in this order, we can be sure that we can actually *use* them in this order in our computer program.

In the forwards system, in our example, the *last* variable to be calculated was $E(t)$, the variable on the left-hand side of the last equation before the definition of the *TARGET*. We want to work out the equation to calculate $F\_E(t)$, for all t. Here we find that we face two different conditions, depending on whether $t = T$ or $t < T$. For $t = T$, $E(t)$ affects *TARGET* directly but does not affect any other variables; thus, we have:

$$F\_E(T) = \frac{\partial TARGET}{\partial E(T)} = 1, \tag{103}$$

which follows from differentiating equation 102.

For $t < T$, we look for occurrences of $E(t)$—lagged or unlagged—across the right-hand sides of *all* the equations in the forwards system. (In other words, we are looking for quantities z_j such that $\partial z_j / \partial E(t)$ is nonzero.) There is only one such occurrence here, located in equation 98; therefore, the sum on the right-hand side of 101 has one nonzero entry in this case. Substituting into equation 101, we get:

$$F\_E(t) = \frac{\partial TARGET}{\partial E(t)} + F\_E(t+1) * \frac{\partial E(t+1)}{\partial E(t)} \tag{104}$$

$$= 0 + F\_E(t+1) * 1 = F\_E(t+1).$$

To simplify our computer program, we can combine 103 and 104 to get:

$$F\_E(t) = 1. \tag{105}$$

Now that we have calculated the feedback to $E(t)$, which was calculated in equation 98, we go back to the previous equation, 97. We try to work out the equation for $F\_Y_3(t)$. Looking across the entire system again, we see the Y_3 appears on the right in only one equation, equation 98. Thus, we deduce:

$$F\_Y_3(t) = \frac{\partial TARGET}{\partial Y_3(t)} + F\_E(t) * \frac{\partial E(t)}{\partial Y_3(t)} \tag{106}$$

$$= 0 + 1 * (2(Y_3(t) - Y_3^*(t))) = 2(Y_3(t) - Y_3^*(t)).$$

Then we go back to calculating the feedback to $Y_2(t)$, which appears *twice* in equation 97. For the case $t < T$, we derive:

$$F\_Y_2(t) = \frac{\partial TARGET}{\partial Y_2(t)} + F\_Y_3(t+1) * \frac{\partial Y_3(t+1)}{\partial Y_2(t)} + F\_Y_3(t) * \frac{\partial Y_3(t)}{\partial Y_2(t)} \tag{107}$$

$$= 0 + F\_Y_3(t+1) * (4Y_2(t+1)) + F\_Y_3(t) * (4Y_2(t-1)).$$

$$= 4F\_Y_3(t+1) * Y_2(t+1) + 4F\_Y_3(t) * Y_2(t-1).$$

Note that $F\_Y_3(t)$ was complicated enough that there is no value in substituting in from equation 106; it is more efficient for the computer to calculate equation 106 as such, and then move on to 107. Also, equation 15 can be used for the case $t = T$ as well, if we adopt the convention that $F\_Y_3(T + 1) = 0$.

Finally, we can calculate the derivatives that we are really interested in. For the weights, which appear only in equation 96, but which affect Y_2 at *all* times t, the chain rule for ordered derivatives yields:

$$F\_W_1 = \sum_{t=1}^{T} F\_Y_2(t) * Y_1(t) \tag{108}$$

$$F\_W_2 = \sum_{t=3}^{T} F\_Y_2(t) * Y_1(t-2). \tag{109}$$

Likewise, for the inputs $Y_1(t)$, we get:

$$F\_Y_1(t) = F\_Y_2(t) * W_1 + F\_Y_2(t+2) * W_2. \quad + \underline{F\_Y_3(t+1) * 3} \tag{110}$$

Equations 106 through 110, together, form a well-ordered backwards system, which we can easily program directly into a computer program or even into a spreadsheet (as in [4]). When debugging complex systems, it often pays to identify select out simple test problems (after testing the big system), and to implement them in a spreadsheet. The computer program to calculate the derivatives would look something like:

Initialize F_W_1, F_W_2, and quantities for $t > T$ to zero
DO FOR $t = T$ to $t = 1$ by –1:
    Invoke equation 106
    Invoke equation 107
    Invoke equation 110
    Set F_W_1 to $F\_W_1 + F\_Y_2(t) * Y_1(t)$
    Set F_W_2 to $F\_W_2 + F\_Y_2(t+2) * Y_1(t)$
    End

For very large, nonmodular forwards systems (a hundred equations or more), it is sometimes inconvenient to have to search through the entire forwards system all over again for each variable. In that case, it may be convenient to write "*F_varN* =" at the front of a blank line, "*F_varN* – 1 =" under that (using variable names instead of numbers), until you have a list, going over several pages, in reverse order. Then you can work your way through each occurrence of each variable as you come to it, adding the appropriate term to the right feedback equation. This kind of inverse search would also allow more efficient automation of this procedure. If you have any doubts about the details, you can use the example above as a test.

10.6.3. How to Construct Dual or Adjoint Subroutines or Functions

Complex systems for neurocontrol or system identification can be very difficult to keep track of if one attempts to write out every equation in the entire system as one big mass of equations. The resulting computer implementation would be even harder to maintain, understand, improve, and debug. On the other hand, the use of modular design—both in research and in implementation—can make these systems quite simple.

A vital part of these designs is the ability to backpropagate *through* different modules. For example, we may have a module that calculates:

$$z_{m+1} = f_{m+1}(z_1, \ldots, z_m) \tag{111}$$

$$z_{N-1} = f_{N-1}(z_1, \ldots, z_{N-2})$$

but does *not* take the final step of calculating *TARGET*, z_N. We assume that the final *TARGET* will be calculated somewhere else. The module here will simply output z_{m+1} through z_{N-1}. In order to backpropagate through this module, we must assume that some *other* module will provide the quantities:

$$F0\_z_i \triangleq \frac{\partial TARGET}{\partial z_i} \qquad m+1 \le i \le N-1. \tag{112}$$

(In actuality, these derivatives may themselves be calculated by backpropagating through some other module.) We must program a *dual* module that *inputs* $F0\_z_i$ and implements:

$$F\_z_i = F0\_z_i + \sum_{j=i+1}^{N-1} F\_z_j * \frac{\partial z_j}{\partial z_i}. \tag{113}$$

To work out what these equations are for any particular ordered system, we can use exactly the same procedures we used in section 10.6.2. Once again, this entire procedure could easily be automated. This would permit control theorists (and modelers of all kinds) to take advantage of backpropagation and the designs of neurocontrol without being restricted to the case of ANNs.

10.7. GRADIENT AND HESSIAN CALCULATIONS FOR SECTION 10.3

The rapid, parallel implementation of the methods of section 10.3 would depend on a rapid calculation of the gradient and (if possible) the Hessian of H, as defined in equation 20, holding the Lagrange multipliers constant. These calculations can be derived by using the chain rule for ordered derivatives. To do this (following section 10.6), we must first specify the forwards system used to calculate the target, H. For simplicity, we will use k as the "time" index (representing $t + k$), use φ instead of F_2, and use p instead of pH. We will use the notation $s'(x)$ for the derivatives of the sigmoid function. (Actually, $s'(x) = s(x)(1 - s(x))$.)

The forwards system can be written as the following, for $k = 0$ to 30:

$$v_j^-(k) = W_{j0}^- + W_{j1}^- p(k-1) + W_{j2}^- p(k-2) + W_{j3}^- p(k-3) \tag{114}$$
$$+ W_{j4}^- \varphi(k) + W_{j5}^- \varphi(k-1) + W_{j6}^- \varphi(k-2) + W_{j7}^- \varphi(k-3)$$

$$x_j^-(k) = s(v_j^-(k)) \tag{115}$$

$$v_1^+(k) = W_{10}^+ + \sum_{j=1}^{h} W_{1j}^+ x_j^-(k) \tag{116}$$

$$p(k) = bias + s(v_1^+(k)) \tag{117}$$

$$U(k) = U(k-1) + (p(k) - p^*)^2 + \lambda_k^{(1)}(p(k) - p_{\min})$$
$$+ \lambda_k^{(2)}(p_{\max} - p(k)) + \lambda_k^{(3)}(\varphi(k) - \varphi_{\min}) + \lambda_k^{(4)}(\varphi_{\max} - \varphi(k))$$

where $U(-1) = 0$ and the target (H) is $U(30)$.

A direct application of the chain rule for ordered derivatives (section 10.6) yields:

$$F\_p(k) = 2(p(k) - p^*) + \lambda_k^{(1)} - \lambda_k^{(2)} \tag{118}$$
$$+ \sum_{j=1}^{h} (F\_v_j^-(k+1) * W_{j1}^- + F\_v_j^-(k+2) * W_{j2}^- + F\_v_j^-(k+3) * W_{j3}^-)$$

$$F\_v_1^+(k) = F\_p(k) * s'(v_1^+(k)) \tag{119}$$

$$F\_x_j^-(k) = F\_v_1^+(k) * W_{1j}^+ \tag{120}$$

$$F\_v_1^-(k) = F\_x_j^-(k) * s'(v_1^-(k)) \tag{121}$$

$$F\_\varphi(k) = \sum_{j=1}^{h} (F\_v_j^-(k) * W_{j4}^- + F\_v_j^-(k+1) * W_{j5}^- \tag{122}$$
$$+ F\_v_j^-(k+2) * W_{j6}^- + F\_v_j^-(k+3) * W_{j7}^-)$$

where we again adopt the convention that *F_anything* (k) equals zero for $k > 30$. Equation 122 yields the desired gradient (i.e., it yields $F\_\phi(k)$ for k from 0 to 30). As in section 10.6, the backwards equations must be invoked backwards in time (i.e., from $k = 30$ back to $k = 0$). To explicitly implement the additional requirement (dynamic equation) that $\phi(k + 7) = \phi(7)$, we could simply modify equation 122 to:

$$F\_\varphi(k) = c(k) * F\_\varphi(k+1) + \sum_j (F\_v_j^-(k) * W_{j4}^- + F\_v_j^-(k+1) * W_{j5}^- + F\_v_j^-(k+2) * W_{j6}^- + F\_v_j^-(k+3) * W_{j7}^-), \tag{123}$$

where we define $c(k)$ as 1 for $k > 6$ and 0 for $k < 7$. Again, it is easy to generalize this procedure for more complex models.

To calculate the Hessian is substantially trickier. We cannot calculate the entire Hessian in *one* pass; however, we *can* calculate the second derivatives with respect to $\phi(\iota)$ and $\phi(k)$, for a *particular* ι and *all* k, in one pass. By repeating this process for all ι, we can get the Hessian at minimum cost.

To generate these equations, we must treat the *entire* system, from equation 114 to equation 122, as an *ordered* system for calculating the target $F\_\phi(\iota)$. We must "forget" that quantities beginning with "$F\_$" have anything to do with derivatives. To avoid confusion, we must use "$G\_$" to represent ordered derivatives of $F\_\phi(\iota)$ with respect to other quantities in that ordered system. We must keep in mind that the order of calculation is from $k = 0$ to $k = 30$ for the forwards block and $k = T$ to $k = \iota$ for the backwards block. Working back through that system, we arrive at the following equations (which have not been implemented or tested as yet):

$$G\_F\_\varphi(k) = 1 \text{ (\textit{for } } k = \iota) \text{ or } 0 \text{ (\textit{otherwise})} \tag{124}$$

$$G\_F\_v_j^-(k) = G\_F\_\varphi(k) * W_{j4}^- + G\_F\_\varphi(k-1) * W_{j5}^- + G\_F\_\varphi(k-2) * W_{j6}^- + G\_F\_\varphi(k-3) * W_{j7}^- \tag{125}$$

$$G\_F\_x_j^-(k) = G\_F\_v_j^-(k) * s'(v_j^-(k)) \tag{126}$$

$$G\_F\_v_i^+(k) = \sum_j G\_F\_x_j^-(k) * W_{ij} \tag{127}$$

$$G\_F\_p(k) = G\_F\_v_i^+(k) * s'(v_i^+(k)) \tag{128}$$

$$G\_F\_U(k) = 0$$

$$G\_p(k) = 2 * G\_F\_p(k) + \sum_{j=1}^{h} (G\_v_j^-(k+1) * W_{j1}^- + G\_v_j^-(k+2) * W_{j2}^- + G\_v_j^-(k+3) * W_{j3}^-) \tag{129}$$

$$G\_v_i^+(k) = G\_F\_v_i^+(k) * F\_p(k) * s''(v_i^+(k)) + G\_p(k) * s'(v_i^+(k)) \tag{130}$$

$$G\_x_j^-(k) = G\_v_i^+(k) * W_{ij}^+ \tag{131}$$

$$G\_v_j^-(k) = G\_F\_v_j^-(k) * F\_x_j^-(k) * s''(v_j^-(k)) + G\_x_j^-(k) * s'(v_j^-(k)) \tag{132}$$

$$G\_\varphi(k) = \sum_{j=1}^{h} (G\_v_j^-(k) * W_{j4} + G\_v_j^-(k+1) * W_{j5} + G\_v_j^-(k+2) * W_{j6} + G\_v_j^-(k+3) * W_{j7}). \tag{133}$$

Of course, equation 133 yields the relevant second derivatives. Once again, multiple passes—for different ι—are needed in this case, because we are looking for an entire matrix of second derivatives. In coding these equations, one first loops forward from $k = \iota$ to $k = T$ in invoking equations 124 through 128 as a block; then one invokes through 129 through 133 for $k = T$ back to $k = 0$. At least for large models, these calculations for the Hessian are an order of N less expensive than those originally proposed by Saint-Donat et al. [1].

In other applications, where we only need one vector based on second derivatives, we can usually get by with a single pass through the system, if we define the appropriate scalar *TARGET* as a function of first derivatives. An example is given in [21].

10.8. IMPLEMENTATION OF MODELS AND METHODS IN SECTION 10.4

Section 10.4 has already given the concepts behind these methods and their implementation; however, it left out the details of the derivative calculations, which will be given here. (For the real-time version of this, see section 10.4.6.3.)

When debugging these programs, the user is urged to print out (or store) the following three items, at a minimum, in each iteration: (1) the change in error from iteration k to iteration $k + 1$; (2) the *prediction* of that change, based on the dot product of the weight change with the gradient calculated in iteration k; (3) the prediction of the same quantity, based on the gradient calculated in iteration $k + 1$. Normally, items 1 and 2 are negative, and 1 lies between 2 and 3. If 1 is not between 2 and 3 (except for a few atypical cases like the first iteration), suspect a bad gradient; test the gradient as described in section 10.6. If 1 is positive, your learning rate is too high; you need to rescale your variables or improve your learning-rate algorithm. If 1 is always negative, but you are making little progress, you need a faster learning rate or you need to consider some of the tricks cited in this book to speed things up.

10.8.1. The Compromise and Pure Robust Methods for Feedforward ANNs or NARX Models

For the sake of brevity and generality, we will use the "modular" approach of section 10.6.3. We will assume a model of order 3, just to make the equations more readable; the generalization to arbitrary order is trivial.

Our first job is to spell out the calculations used to obtain the loss function for a given set of parameters; from equations 63 through 65, we may write:

$$\hat{\mathbf{Y}}(t) = f(\mathbf{Y}(t-1), \mathbf{Y}(t-2), \mathbf{Y}(t-3), \boldsymbol{u}(t), W) \tag{134}$$

$$\bar{Y}_i(t) = (1 - w_i)\hat{\bar{Y}}_i(t) + w_i Y_i(t) \qquad i = 1, n \tag{135}$$

$$E = \sum_t L(\hat{\bar{Y}}(t), Y(t), w). \tag{136}$$

To implement the pure robust method, simply set $w_i = 0$, use the *same* derivative equations as given below, but skip the equations used only to adapt w_i. In equation 134, the parameters or weights of the model—W—are described as a vector, just for convenience; there is no assumption that the number of parameters is in any way related to the number of variables in the model. The vector $u(t)$ represents all exogenous variables *used* to predict $Y(t)$, without regard to their actual source in space and time. The loss function L could be any of the various loss functions described in section 10.4.

Using the procedures of section 10.6, we may immediately deduce:

$$F\_\hat{\bar{Y}}_i(t) = \frac{\partial}{\partial \hat{\bar{Y}}_i(t)} L(\hat{\bar{Y}}(t), Y(t), W) \tag{137}$$

$$\begin{aligned} F\_\bar{Y}(t) = {} & F1\_f(\bar{Y}(t), \bar{Y}(t-1), \bar{Y}(t-2), u(t+1), W, F\_\hat{\bar{Y}}(t+1) \\ & + F2\_f(\bar{Y}(t+1), \bar{Y}(t), \bar{Y}(t-1), u(t+2), W, F\_\hat{\bar{Y}}(t+2)) \\ & + F3\_f(\bar{Y}(t+2), \bar{Y}(t+1), \bar{Y}(t), u(t+3), W, F\_\hat{\bar{Y}}(t+3)) \end{aligned} \tag{138}$$

$$F\_W = \sum_t FW\_f(\bar{Y}(t), \bar{Y}(t-1), \bar{Y}(t-2), u(t), W, F\_\hat{\bar{Y}}(t)) \tag{139}$$

$$F\_w_i = \sum_t F\_\bar{Y}_i(t) * (Y_i(t) - \hat{\bar{Y}}_i(t)). \tag{140}$$

The required derivatives are calculated by equations 139 and 140. In equations 138 and 139, we assume that you have coded up the dual subroutine, *F_f*, for the function *f*. We have used the abbreviation "*F1_*" (instead of $F_{Y(t-1)}\_f$) for the subroutine which calculates derivatives with respect to the *first* vector argument of *f*, and likewise for *F2* and *F3*, just to avoid possible confusion about time lags.

In practice, it is easier to calculate the sums in equations 138 and 139 as *running* sums (as with the weight derivatives), in computer programming. This yields code that begins by initializing many things to zero, followed by the main loop:

DO FOR t = T to t = 1 by −1:
  Invoke equation 137;
  FOR k = 1 to k = 3: F_$\bar{Y}$(t − k) = F_$\bar{Y}$(t − k) + Fk_f(...,F_$\hat{\bar{Y}}$(t));
  F_W = F_W + FW_f(..., F_$\hat{\bar{Y}}$(t))
  FOR i = 1 to i = n: F_w_i = F_w_i + F_$Y_i(t+k)$ * ($Y_i(t+k)$ − $\hat{\bar{Y}}_i(t+k)$);
  End

The next to last instruction uses "$t + k$" instead of "t," to make sure that F_Y_i has been completely added up; therefore, after the core loop, one needs a quick loop for $k = 1$ to $k = 3$ to add in the final components of F_w_i. The cost of running code like this will be far less if you actually use a *single* subroutine, with multiple outputs, to provide all the derivatives based on $\boldsymbol{F\_\hat{Y}}(t)$.

There is one more detail of relevance to computer programmers. The dual subroutine usually requires knowledge of the *intermediate* variables of the original model f. This can be handled either by *storing* all the intermediate information on the initial forwards pass, or by *recalculating* them from the inputs of the model (which need to be stored in any case). The tradeoff here is very application-dependent.

10.8.2. Classical Adaptation of Time-lagged Recurrent Networks (TLRNs)

For a classical TLRN with classical maximum likelihood training, the calculation of error is based on equations 53, 54, 55, 56, and 22, in that order. If we apply the chain rule for ordered derivatives (section 10.6) to that ordered system, we immediately arrive at the following equations to calculate the derivatives we need:

$$F\_\hat{Y}_i(t) = \hat{Y}_i(t) - Y_i(t) \qquad 1 \le i \le n \tag{141}$$

$$F\_x_i(t) = F\_\hat{Y}_{i-N}(t) + \sum_{j=i+1}^{N+n} F\_v_j(t) W_{ji} + \sum_{j=m+1}^{N+n} F\_v_j(t+1) W'_{ji} \qquad m < i \le N+n \tag{142}$$

$$F\_v_i(t) = F\_x_i(t) * s'(v_i(t)) \qquad m < i \le N+n \tag{143}$$

$$F\_W_{ij} = \sum_t F\_v_i(t) * x_j(t) \tag{144}$$

$$F\_W'_{ij} = \sum_t F\_v_i(t) * x_j(t-1) \tag{145}$$

where we adopt the conventions that $F\_\hat{Y}_k$ is zero for $k < 1$ and that $F\_anything(t)$ is zero for $t > T$. The generalization for longer time lags is trivial [22].

For a sticky TLRN with classical training, the calculation of error is based on equations 53, 57, 55, 56, and 22, in that order. The derivative equations are:

$$F\_\hat{Y}_i(t) = \hat{Y}_i(t) - Y_i(t) \tag{146}$$

$$F\_x_i(t) = F\_\hat{Y}_{i-n}(t) + \sum_{j=i+1}^{N+n} F\_v_j(t) * W_{ji} \tag{147}$$

$$F\_v_i(t) = F\_x_i(t) * s'(v_i(t)) + \sum_{j=m+1}^{N+n} F\_v_j(t+1) * W_{ji}' \tag{148}$$

$$F\_W_{ij} = \sum_t F\_v_i(t) * x_j(t) \tag{149}$$

$$F\_W_{ij}' = \sum_t F\_v_i(t) * v_j(t-1) \tag{150}$$

The case of a mixed system, with a combination of classical, sticky, and extreme sticky neurons can again be handled by a straightforward use of the procedures in section 10.6.

10.9. THE COMPROMISE METHOD FOR RECURRENT MODELS (ANN OR OTHER)

As in section 10.8.1, we will use a modular approach here.

The notation here is somewhat tricky, since we need to consider a forwards model which outputs *two* vectors—a prediction, $\hat{\boldsymbol{Y}}(t)$, and a filtered state vector, $\boldsymbol{R}(t)$. In section 10.8.2, we assumed that the vector $\boldsymbol{R}(t)$ was made up of $x_{m+1}(t)$ through $x_{N+n}(t)$ or $v_{m+1}(t)$ through $v_{N+n}(t)$, though in most applications we restrict the recurrence to a subset of the neurons. For simplicity here, I will assume a model that is only second-order in the predicted variables and first-order in the recurrent variables. We may write the forwards system used to evaluate error as:

$$\hat{\boldsymbol{Y}}(t) = f_1(\bar{\boldsymbol{Y}}(t-1), \bar{\boldsymbol{Y}}(t-2), \boldsymbol{R}(t-1), \boldsymbol{u}(t), \boldsymbol{W}) \tag{151}$$

$$\boldsymbol{R}(t) = f_1(\bar{\boldsymbol{Y}}(t-1), \bar{\boldsymbol{Y}}(t-2), \boldsymbol{R}(t-1), \boldsymbol{u}(t), \boldsymbol{W}) \tag{152}$$

$$\bar{Y}_i(t) = (1 - w_i)\hat{Y}_i(t) + w_i Y_i(t) \tag{153}$$

$$E = \sum_t L(\hat{\boldsymbol{Y}}(t), \boldsymbol{Y}(t), \boldsymbol{W}). \tag{154}$$

If we apply the chain rule for ordered derivatives, as we did in section 10.8.1, we end up with equation 137 followed by:

$$\begin{aligned} F\_\bar{\boldsymbol{Y}}(t) = {} & F1\_f(\bar{\boldsymbol{Y}}(t), \bar{\boldsymbol{Y}}(t-1), \boldsymbol{R}(t), \boldsymbol{u}(t+1), \boldsymbol{W}, F\_\hat{\boldsymbol{Y}}(t+1), F\_\boldsymbol{R}(t+1)) \\ & + F2\_f(\bar{\boldsymbol{Y}}(t+1), \bar{\boldsymbol{Y}}(t), \boldsymbol{R}(t+1), \boldsymbol{u}(t+2), \boldsymbol{W}, F\_\hat{\boldsymbol{Y}}(t+2), F\_\boldsymbol{R}(t+2)) \end{aligned} \tag{155}$$

$$F\_\boldsymbol{R}(t) = F3\_f(\bar{\boldsymbol{Y}}(t), \bar{\boldsymbol{Y}}(t-1), \boldsymbol{R}(t), \boldsymbol{u}(t+1), \boldsymbol{W}, F\_\hat{\boldsymbol{Y}}(t+1), F\_\boldsymbol{R}(t+1)) \tag{156}$$

$$F\_\boldsymbol{W} = \sum_t FW\_f(\bar{\boldsymbol{Y}}(t-1), \bar{\boldsymbol{Y}}(t-2), \boldsymbol{R}(t-1), \boldsymbol{u}(t), \boldsymbol{W}, F\_\hat{\boldsymbol{Y}}(t), F\_\boldsymbol{R}(t)) \tag{157}$$

followed by equation 140. The key thing about these equations is that there was really only *one* forwards model f with *two outputs*; if we have $F\_$ derivatives available for both of these outputs, we can use a *single* dual subroutine to calculate the derivatives with respect to all four of the relevant vector arguments of f.

Clearly, the main difficulty in implementing equations 155 through 10.8.24 is the need to program the dual subroutine, based on the methods of section 10.6. Perhaps an example might help. Suppose that the recurrent system under discussion is a classical TLRN. In that case, the "model" may take the form:

$$x_i(t) = \bar{Y}_i(t-1) \qquad 1 \le i \le n \tag{158}$$

$$x_{i+n}(t) = \bar{Y}_i(t-2) \qquad 1 \le i \le n \tag{159}$$

$$x_{i+2n}(t) = R_i(t-1) \qquad 1 \le i \le r \tag{160}$$

$$x_{i+2n+r}(t) = u_i(t) \qquad 1 \le i \le m-2n-r \tag{161}$$

$$v_i(t) = \sum_{j=1}^{i-1} W_{ij} x_j(t) \qquad m < i \le N+n \tag{162}$$

$$x_i(t) = s(v_i(t)) \qquad m < i \le N+n \tag{163}$$

$$\hat{Y}_i(t) = x_{N+i}(t) \qquad 1 \le i \le N+n \tag{164}$$

$$R_i(t) = x_{N-r+i}(t) \qquad 1 \le i \le r. \tag{165}$$

Equations 158 through 165 can be coded into *one subroutine* that produces two outputs—$\boldsymbol{Y}(t)$ and $\boldsymbol{R}(t)$. Applying the methods of section 10.6, we arrive at the equations for the dual subroutine:

$$F\_x_i(t) = \sum_{j>i} F\_v_j(t) * W_{ji} + F\_\hat{Y}_{i-N}(t) + F\_R_{i-N+r}(t) \tag{166}$$

$$F\_v_i(t) = F\_x_i(t) * s'(v_i(t)) \tag{167}$$

$$F\_W_{ij}(t) = F\_v_i(t) * x_j(t) \tag{168}$$

$$F\_R_i(t-1) = F\_x_{i+2n}(t) \tag{169}$$

$$F\_\bar{Y}_i(t-2) = F\_x_{i+n}(t) \qquad (F2\_) \tag{170}$$

$$F\_\bar{Y}_i(t-1) = F\_x_i(t) \qquad (F1\_). \tag{171}$$

Equations 166 through 171 can be coded into a *single* dual subroutine that yields all four $F\_$ arrays needed in equations 155 through 157. In equation 166, as in earlier examples, we treat $F\_\hat{Y}_i$ and F_R_i as zero for subscripts that are out of bounds.

Once again, equations 151 through 154 can be used with *any* model $\boldsymbol{f}$, so long as you know how to program the dual subroutine for $\boldsymbol{f}$. For example, $\boldsymbol{f}$ could be a simultaneous-recurrent neural network or some other recurrent structure (such as a fuzzy inference structure or a finite element code or a simultaneous model from econometrics); to program a dual subroutine for such an $\boldsymbol{f}$, see Chapter 13. The insertion of such an $\boldsymbol{f}$ into this system allows one to build a single modeling system that combines together the advantages of simultaneous recurrence, time-lagged recurrence, and robust estimation; once again, we believe that the higher centers of the brain rely on a similar combination (along with real-time learning) [66,67].

10.10 REFERENCES

[1] J. Saint-Donat, N. Bhat, and T. J.McAvoy, Neural net based model predictive control. *Int. J. Control*, in press.

[2] N. Bhat and T. McAvoy, Information retrieval from a stripped backpropagation network. In *AICheE Proceedings*, 1990.

[3] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1974.

[4] P. J. Werbos, Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods. *IEEE Trans. Systems, Man & Cybernetics*, March/April 1989.

[5] M. J. Piovoso et al., Neural network process control. In *Proceedings: Analysis of Neural Networks Applications Conference*. ACM Order #604911. New York: ACM, 1991.

[6] T. J. McAvoy et al., Interpreting biosensor data via backpropagation. *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, IEEE, New York, 1989.

[7] N. Bhat and T. McAvoy, Use of neural nets for dynamic modeling and control of chemical process systems. *1989 American Control Conference*; *Computers and Chemical Engineering*, in press.

[8] T. McAvoy, Y. Arkun, and E. Zafiriou, eds., *Model-Based Process Control: Proceedings of the 1988 IFAC Workshop*, Pergamon press, Oxford, U.K., 1989.

[9] C. Gingrich, K. Kuespert, and T. McAvoy, Modeling human operators using neural networks. In *Proceedings of Instrument Society of America Conference*, October 1990.

[10] D. Kuespat and T. McAvoy, Neural network behavior analysis of human process operators. In *AIChE Proceedings*, 1991.

[11] G. Birky and T. J. McAvoy, A neural net to learn the design of distillation controls, preprints of IFAC DYCORD Symposium, pp.147–153, Maastricht, Netherlands, August 1989.

[12] S. Naidu, E. Zafirou, and T. McAvoy, Application of neural networks with detection of sensor failure during the operation of a control system. *IEEE Control Systems,* 10:49–55, April 1990.

[13] W. Cary et al., Chemometric analysis of multisensor arrays. *Sensors and Actuators,* 1:223–234, 1986.

[14] T. Tanaka et al., A trouble forecasting system by a multi-neural network on a continuous casting process of steel production. *Artificial Neural Networks, Vol. 1,* T. Kohonen et al., eds., North-Holland Amsterdam, 1991.

[15] P. J. Werbos, Neurocontrol and fuzzy logic. *International Journal of Approximate Reasoning,* February 1992. An earlier, less complete version appeared in *Proceedings of the Second Joint Technology Workshop on Neural Networks and Fuzzy Logic (1990), Vol. II,* NASA Conference Publication 10061. See also P. Werbos, Elastic fuzzy logic: a better fit to Neurocontrol. *I:zuka-92 Proceedings,* Japan, 1992.

[16] P. J. Werbos, Making diagnostics work in the real world. *Handbook of Neural Computing Applications*m, A. Maren, ed., , pp.337–338, Academic Press, New York, 1990.

[17] E. Blum, Approximation theory and feedforward networks, *Neural Networks,* 4, 1991.

[18] G. Cybenko, Approximation by superpositions of sigmoidal functions. *Mathematics of Controls, Signals and Systems,* 2:303–314, 1988.

[19] E. Sontag, Feedback stabilization using two-hidden-layer nets. SYCON-90-11. Rutgers University Center for Systems and Control, New Brunswick, NJ, October 1990.

[20] P. J. Werbos, Applications of advances in nonlinear sensitivity analysis. *Systems Modeling and Optimization: Proceedings of the 1981 IFIP Conference,* R. Drenick and F. Kozin, eds., Springer-Verlag, New York, 1982.

[21] P. J. Werbos, Backpropagation: Past and future. *ICNN Proceedings.* IEEE, New York, 1988. A transcript of the talk, with slides, is somewhat more readable and is available from the author.

[22] P. J. Werbos, Backpropagation through time: What it is and how to do it. *Proceedings of the IEEE,* October 1990.

[23] D. Rumelhart, G. Hinton, and R. Williams, *Parallel Distributed Processing,* MIT Press, Cambridge, MA, 1986.

[24] D. F. Shanno, Recent advances in numerical techniques for large-scale optimization. *Neural Networks for Control*, W. Miller, R. Sutton, and P. J. Werbos, eds., MIT Press, Cambridge, MA, 1990.

[25] See also Chapter 13.

[26] E. R. Panier and A. L. Tits, *On Feasibility, Descent and Superlinear Convergence in Inequality Constrained Optimization*. Systems Research Center, University of Maryland, College Park, MD, 1989.

[27] J. Zhou and A. L. Tits, *User's Guide for FSQP*, Systems Research Center, University of Maryland, College Park, MD, 1989.

[28] P. J. Werbos, A menu of designs for reinforcement learning. In W. Miller, R. Sutton, and P. J. Werbos, *op. cit.* [24].

[29] T. Su, T. J. McAvoy, and P. J. Werbos, Identification of chemical processes via parallel training of neural networks. Submitted to *IECE*, 1991.

[30] C. F. Mosteller and R. E. Rourke, *Sturdy Statistics: Nonparametric and Order Statistics,* Addison-Wesley, Reading, MA, 1973.

[31] L. Ljung, *System Identification, Theory for the User*, Prentice-Hall, Englewood Cliffs, NJ, 1987

[32] P. Young, Parameter estimation for continuous-time models: a summary. *Automatica,* 17:(1), 1981.

[33] K. S. Narendra and K. Parthasarathy, Identification and control of dynamic systems using neural networks. *IEEE Trans. Neural Networks*, 1, 1990.

[34] P. J. Werbos, Learning how the world works: Specifications for predictive networks in robots and brains. In *Proceedings of the SMC Conference.* IEEE, New York, 1987.

[35] P. J. Werbos, Neurocontrol and related techniques. In A. Maren, *op. cit.* [16].

[36] P. J. Werbos, Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 1977.

[37] P. J. Werbos and J. Titus, An empirical test of new forecasting methods derived from a theory of intelligence: the prediction of conflict in Latin America. *IEEE Trans. Systems, Man & Cybernetics*, September 1978.

[38] M. Kawato, Computational schemes and neural network models for formation and control of multijoint arm trajectory. In W. Miller, R. Sutton, and P. J. Werbos, *op.cit.* [24].

[39] B. Widrow and M. E. Hoff, Adaptive switching circuits. *WESCON Convention Record, IV*, pp. 96–104. Reprinted in *WESCON/89*.

[40] B. A. Pearlmutter, Dynamic recurrent neural networks. Technical report CMU-CS-90-196. Carnegie-Mellon University, Pittsburgh, PA, 1990.

[41] N. Baba, A new approach for finding the global minimum of error function of neural networks. *Neural Networks*, 5:367–373, 1989.

[42] F. J. Solis and J. B. Wets, Minimization by random search techniques. *Mathematics of Operations Research*, 6:19–30, 1981.

[43] M. J. Willis et al., On artificial neural networks in process engineering. *IEEE*, Part D, 1991.

[44] H. J. Bremermann and R. W. Anderson, An alternative to backpropagation: a simple rule for synaptic modification for neural net training and theory. Department of Mathematics, University of California at Berkeley, 1989.

[45] K. M. Pedersen, M. Kummel, and H. Soeberg, Monitoring and control of biological removal of phosphorus and nitrogen by flow injection analyzers in a municipal pilot scale wastewater treatment plant. Submitted to *Analytical Cim. Acta*, 1990.

[46] J. H. Gary and G. E. Handwerk, *Petroleum Refining: Technology and Economics*. Marcel Dekker, New York, 1975.

[47] R. J. Williams and J. Peng, An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 4, 1990.

[48] H. Koivisto, *Minimum Prediction Error Neural Controller*, Department of Electrical Engineering, Tampere University of Technology, Tampere, Finland, 1990.

[49] D. Nguyen, *Applications on Neural Networks in Adaptive Control*, Ph.D. thesis. Stanford University, Palo Alto, CA, June 1991.

[50] G. Box and G. Jenkins, *Time-Series Analysis: Forecasting and Control*, Holden-Day, San Francisco, CA, 1970.

[51] I. J. Leontaritis and S. A. Billings, Input-output parametric models for nonlinear systems. *Int. J. Control*, 41:303–344, 1985.

[52] A. E. Bryson and Y. C. Ho, *Applied Optimal Control*, Ginn and Co., Waltham, MA, 1969.

[53] A. Gelb, ed., *Applied Optimal Estimation*. MIT Press, Cambridge, MA, 1974.

[54] P. J. Werbos, Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:(4) (submitted August 1987).

[55] P. J. Werbos, Energy and population: Transitional issues and eventual limits. *The Elephants in the Volkswagen: Facing the Tough Questions About Our Crowded Nation*, L.Grant, ed., W. H. Freeman, New York, 1992.

[56] P. J. Werbos, Econometric techniques: Theory versus practice. *Energy: The International Journal*, March/April 1990. (Also forthcoming in a Pergamon book edited by Weyant and Kuczmowski.)

[57] R. J. Solomonoff, A formal theory of inductive inference. *Information and Control*, March/June 1964.

[58] N. Tishby, E. Levin, and S. Solla, Consistent inference of probabilities in layered networks: Predictions and generalization. In *IJCNN Proceedings, Vol. II*, IEEE, New York, 1989.

[59] P. Papantoni-Kazakos and D. Kazakos, Fundamental neural structures, operations and asymptotic performance criteria in decentralized binary hypothesis testing. In *Proceedings of the IEEE Conference on Neural Networks for Ocean Engineering*, IEEE No. 91CH3064-3. IEEE, New York, 1991.

[60] A. J. Dempster, *J. American Statistical Assoc.*, 72:(357), March 1977.

[61] P. J. Werbos and J. Titus, *Predicting Conflict in Latin America: The First Full Test of Werbos's Robust Method*. Final report of DARPA project funded through ONR contract No. N00014-75-C-0846. This document was submitted in January 1978 to DARPA, but never entered into the defense documentation system.

[62] H. Hjalmarsson and L. Ljung, How to estimate model uncertainty in the case of undermodelling. *1990 American Control Conference*, IEEE, New York, 1990.

[63] G. C. Goodwin and R. L. Payne, *Dynamic System Identification: Experiment Design and Data Analysis*, Academic Press, New York, 1977.

[64] K. Niida, J. Tani, and I. Koshijima, Application of neural networks to rule extraction for operator data. *Computer and Chemical Engineering*, submitted in 1990.

[65] A. R. Barron, Predicted square error: a criterion for automatic model selection. *Self-Organizing Methods in Modeling*, S. J. Farlow, ed.

[66] P. J. Werbos, The cytoskeleton: Why it may be crucial to human learning and to neurocontrol. *Nanobiology*, 1:(1), 1991 (a new journal from Carfax).

[67] P. J. Werbos, Neurocontrol, biology and the mind. In *IEEE Proceedings SMC*, IEEE, New York, 1991.